

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

非卖品！！严禁（售卖和上传互联网平台）！！

仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

Broadview
www.broadview.com.cn

Redis实战案例书！

Packt

中文版

Redis 4.x Cookbook

黄鹏程 王左非 著

梅隆魁 译



黄东旭

PingCAP CTO和
CodisLabs
联合创始人

顾睿

Redis长期支持者、
贡献者和核心开发者
Redisson团队成员

作序力荐



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

作者介绍

黄鹏程

过去五年多一直在中国民生银行（2017年中国民生银行在世界排名前1000名的银行中位列29位）担任软件工程师及大数据基础设施团队的负责人，负责为整家银行提供大数据基础设施服务。同时，作为这家银行的Redis技术负责人，他将大部分精力投入到了在生产环境中如何更好地使用Redis中。此外，他也是一名Redis贡献者。你可以通过搜索“gnuhpc”在LinkedIn或微信上找到他。

王左非

一名居住在美国旧金山湾区的经验丰富的软件工程师，具有五年多的软件行业经验，曾参与过涉及很多不同技术的项目，目前在爱彼迎（Airbnb）工作。左非乐于学习新事物和分享知识，还喜欢在闲暇之余读书、旅行和捣鼓无线电。

译者简介

梅隆魁

2013年硕士毕业于北京邮电大学计算机科学与技术专业嵌入式系统与网络通信方向。毕业后就职于中国民生银行总行信息科技部，主要从事J2EE企业级及分布式系统的应用和架构设计开发及项目管理工作，业余对嵌入式软硬件、移动应用开发及Android移动安全也有所涉猎，是一名“会画圆”且“能画圆”的工程师。

非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

中文版

Redis 4.x Cookbook

黄鹏程 王左非 著

梅隆魁 译

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

内容简介

Redis 是一个十分热门的内存数据库，号称后端的“瑞士军刀”，它拥有诸多优良特性，已经被越来越多的公司采用，值得每一位开发者学习。通过本书讲述的 Redis 在设计、开发和运维等方面的 80 多个实战案例，读者不仅可以由浅入深地学到有关 Redis 的几乎所有知识，还可以将案例中所讲解的内容直接用于包括设计、开发和运维等在内的各类生产实践。书中的每一个案例、每一个案例中所涉及的各种知识、命令和工具等，均来自作者一线企业级应用的总结；本书中总结的各类参数配置和故障诊断的案例等，也均来自作者真实企业级运维工作的经验。

本书通过可实战的 80 多个案例全面系统地讲解了 Redis 技术应用，适合所有对 Redis 感兴趣的开发与运维人员阅读和参考。

Copyright © 2018 Packt Publishing . First published in the English language under the title *Redis 4.X Cookbook*.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2018-3197

图书在版编目（CIP）数据

Redis 4.x Cookbook 中文版/ 黄鹏程，王左非著；梅隆魁译. —北京：电子工业出版社，2018.5
ISBN 978-7-121-34081-9

I. ①R…II. ①黄…②王…③梅…III. ①数据库—基本知识 IV. ①TP311.138

中国版本图书馆 CIP 数据核字（2018）第 077245 号

责任编辑：孙学瑛 sxy@phei.com.cn

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×1092 1/16 印张：20 字数：456 千字

版 次：2018 年 5 月第 1 版

印 次：2018 年 5 月第 1 次印刷

定 价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819 faq@phei.com.cn。

非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

致谢

献给我的妻子，汪婷，她总是接纳并支持我的各种忙碌、折腾和雄心。你是而且永远是我完美的妻子和我们孩子的母亲。

献给我的父母，你们是天下最好的父母和榜样。

献给我刚刚出生的孩子——好好——你是我们珍爱的小天使。

——黄鹏程

献给我的父母：王宏武和杨九香。

——王左非

非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

我现在还记得我自己第一次听说 Redis 时是多么的惊讶。Redis 是如此的优雅和强大，以至于堪称后端应用的“瑞士军刀”，当然也很快。在现代数据中心中，内存的单位成本正在变得越来越低；因此，Redis 能够在现代应用的存储架构中扮演重要的角色也就没什么让人感到意外的了。坦白地说，Redis 已经远远地超出基于内存的缓存的范畴了。

我是一名数据库工程师。Codis，作为一个分布式的 Redis 中间件，是我的第一个开源项目，已经在社区中被广泛地使用——我对此感到非常自豪。Codis 提供了一种基于代理的方案来解决 Redis 在伸缩性方面的问题，也是 Redis Cluster 的一个替代方案。作为国内最早使用 Redis 和进行 Redis 开发的人员之一，我见证了 Redis 的逐渐流行。也更加感谢 Redis 让我遇见了亲爱的朋友、Redis 专家和 Contributor，黄鹏程。当他告诉我他想写一本关于 Redis 的书时，我就告诉他我一定会买一本。最后，他完成了本书，也让我有了校对本书早期版本并写这个推荐的荣幸。

选择这本书你是一定不会后悔的。Redis 4.0 发布后，引入了许多重大的变化和功能。不管你是新手，还是像我一样对 Redis 有一定开发经验的人，都会从这本书中学到新的技巧。

黄东旭

PingCAP CTO 和 CodisLabs 联合创始人

非卖品！！严禁（售卖和上传互联网平台）！！

仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

推荐语

作为 Redisson 项目的一名贡献者和成员，以及一名长期的 Redis 支持者，我已经见证了 Redis 在从一个版本到另一个版本的迭代过程中稳步地获得了业界的认可。在参加了几个由许多 Redis 和 Redisson 社区成员参与的讨论后，我不禁想到：如果能有一本有关 Redis 最新信息的权威书籍，使得 Redis 用户不用再在 Stack Overflow 上翻来翻去就能找到所遇到问题的答案该多好啊！

很明显，鹏程和左非跟我有相同的想法。

我认识鹏程已经有相当一段时间了。他是 Redis 中国社区一位著名的 Redis 爱好者，管理着一个非常活跃的群，其中的成员包括了 Redis 的核心贡献者、工具/库的作者、Redis 的支持者及日常用户。这本书的内容正是鹏程最擅长的：Redis，或者更准确地说 Redis 4.x。很多人知道，他并不是靠管理这个群为生，他是中国民生银行大数据基础设施的负责人及 Redis 的负责人。他的职责之一是确保作为这家银行基础设施中最重要部分之一的 Redis 被正确地使用和管理。正是他在这家银行的工作经历启发他编写并完成了本书。

本书覆盖的主题十分全面且组织得非常有逻辑，主要面向初学者和中级用户。初学者可以在本书中找到大量有用的示例、图表和指南。中级读者也会很欣喜地发现作者深入地解释了 Redis 的工作原理，并针对每一个主题给出了进一步的阅读建议。

就我个人而言，我非常喜欢针对每一个 Redis 配置选项的详细解释。我认为这本书对我来说同样是一本有用的手册，我会把它放在桌旁作为参考。我希望读者在阅读本书时能够像我一样感到愉快，也希望读者在学到有关 Redis 4.x 的许多优良特性的同时，也能认识到本书的价值。

顾睿

Redis 长期支持者、贡献者和核心开发者，Redisson 团队成员

贡献者

关于作者

黄鹏程过去五年多一直在中国民生银行（2017 年在世界排名前 1000 名的银行中位列第 29 位）担任软件工程师及大数据基础设施团队的负责人，负责为整家银行提供大数据基础设施服务。同时，作为这家银行的 Redis 技术负责人，他将大部分精力投入到了在生产环境中更好地使用 Redis 中。此外，他也是一名 Redis 贡献者。你可以通过搜索“gnuhpc”在 LinkedIn 或者微信上找到他。

我要感谢妻子的支持和鼓励。此外，还要感谢技术评审们提供的有价值的反馈：

Domagoj Katavić (Vectra Networks)

Ihor Malinovskiy (RedisDesktop)

赵昭、付磊、张铁蕾（阿里巴巴）

唐聪（腾讯）

梅路晓（火币）

张海雷（汽车之家）

刘鹏（链家）

吴建超（Oppo）

黄华平（脉脉）

窦锦帅（滴滴）

黄健宏（huangz）

王左非是一名居住在美国旧金山湾区的经验丰富的软件工程师。他有 5 年多的软件行业经验，曾参与过涉及很多不同技术的项目，目前在爱彼迎（Airbnb）工作。左非乐于学习新事物和分享知识，还喜欢在闲暇之余读书、旅行和捣鼓无线电。

关于技术评审

Domagoj Katavic 电子和计算机工程专业硕士毕业，目前在 Vectra Networks 工作（一家从事识别实时网络攻击的网络安全公司）。

此前，他曾在 Planet9 Energy 公司（一家英国的能源供应商）和 Codeanywhere（一家云 IDE 公司）工作。此外，他还是克罗地亚独立大学（University of Split）FESB 的助理教授。

Ihor Malinovskiy 是一位来自乌克兰的编程狂人。他 2009 年刚参加工作时主要从事 Web 开发，之后曾在诸如广告、医疗和云计算等多个不同领域中工作。Ihor 是一名开源爱好者，也是 OpenStack

项目的核心开发者之一。2013 年，他开源了自己的产品——一款名为 Redis Desktop Manager 的 Redis 桌面 GUI。

关于译者

梅隆魁，2013 年硕士毕业于北京邮电大学计算机科学与技术专业嵌入式系统与网络通信方向。毕业后就职中国民生银行总行信息科技部，主要从事 J2EE 企业级及分布式系统的应用和架构设计开发及项目管理工作，业余对嵌入式软硬件、移动应用开发及 Android 移动安全也有所涉猎，是一名“会画圆”且“能画圆”的工程师。

前言

Redis 作为一个流行的 key-value 内存数据存储，由于性能高、数据类型丰富、API 功能强大、可用性高及架构可伸缩等特点，最近受到了越来越多的关注。自 2017 年以来，Redis 已经成功地在 DB-Engine 数据库排行榜（DB-Engine Complete Ranking）中排到了 9/10。在那之前，Redis 甚至还占据过 DB-Engine 键值存储分类榜单的第一名相当长一段时间。从早期的 2.x 版本到最新的 4.x 版本，Redis 引入了很多优秀的特性来帮助希望在业务场景中交付低延迟服务的用户。

《Redis 4.x Cookbook 中文版》基于最新的 Redis 4.x 版本，向读者提供了深入浅出的实战案例和相关的背景知识。本书涵盖了 Redis 的几乎所有方面，从 Redis 基本数据类型，一直到诸如高可用、集群化、管理和故障诊断等高级主题。

基于实践是最好的老师（Learning by doing is the best approach）的理念，本书的作者不遗余力地通过真实的用例向读者呈现有关 Redis 的知识。换句话说，本书为许多常见的开发和维护问题提供了开箱即用的解决方案。即便是在读者的个人计算机上，只要遵循本书操作步骤小节中的步骤，读者都能够很容易地理解每一个实战案例的关键点。此外，仅仅知道如何使用 Redis 达成工作目标是不够的；工作原理小节对读者在某项任务中所执行的步骤进行了解释性的说明。在更多细节小节中，本书还提供了有关 Redis 内部工作原理的相关基本信息和必要的解释。读者越是了解 Redis 的工作原理，就越能对工程中所涉及的权衡作出明智的决定。每一个实战案例都是按照上述的方式组织的。

最后，我们希望本书能让读者更好地了解 Redis，并让读者在自己的场景中使用 Redis 时能够学到更多的最佳实践。

0.1 预期读者

本书面向的是希望开始使用 Redis 或加深对其认知的开发人员、架构师和 DBA。如果读者想使用 Redis 设计高性能、可伸缩的数据库解决方案，那么本书将通过各种各样的实战案例来引领读者全面深入地了解 Redis。本书对于寻求日常运维 Redis 工作中所碰到的常见问题解决方案的 DBA 而言同样有用。本书涵盖了使用 Redis 所涉及的所有方面，并为 Redis 的日常使用提供了全方位的解决方案和技巧提示。尽管要充分利用本书需要对 Redis 有一些基本的理解，但也并不是必需的。

0.2 主要内容

第 1 章，开始使用 Redis，主要涉及 Redis 服务器端的安装和基本操作，包括启动和停止 Redis 服务器、使用 `redis-cli` 连接到 Redis 和获取服务器信息。在本章的最后，还介绍了 Redis 事件模型和 Redis 通信协议。

第2章，数据类型，主要涉及 Redis 的数据类型和操作数据类型的常见 API 命令。本章介绍了 Redis 4.x 版本中支持的所有数据类型（字符串 string、列表 list、哈希 hash、集合 set、有序集合 sorted set、HyperLogLog 和 Geo）。本章还讨论了基本的 Redis 键管理。

第3章，数据特性，主要涉及一些有用的 Redis 特性，这些特性使操作数据变得更加容易。本章首先展示了如何使用位图（bitmap）、SORT 命令和设置键的过期时间。之后，向读者介绍了 Redis 的三个重要功能：管道（pipeline）、事务（transaction）和发布订阅（PubSub）。在本章的最后，我们演示了如何在 Redis 中编写和调试 Lua 脚本。

第4章，使用 Redis 进行开发，演示了如何使用 Redis 开发应用程序。首先，本章讨论了 Redis 的使用场景和数据类型及 API 的选择。之后，本章展示了使用 Redis 客户端库 Jedis 和 redis-py 开发 Java 和 Python 应用程序的示例。最后，本书介绍了在 Spring Framework 中使用 Redis 及在 MapReduce/Spark 作业中使用 Redis 的例子。

第5章，复制（Replication），主要涉及 Redis 的复制机制。本章展示了如何配置 Redis 从实例并解释了 Redis 主从复制的工作原理。然后，本章对 Redis 调优及主从复制相关的故障排除主题进行了讨论。

第6章，持久化（Persistence），介绍了 Redis 中的两种持久化方式：RDB 和 AOF。本章展示了如何在 Redis 中启用 RDB 和 AOF 来实现持久化，并解释了持久化的工作原理。本章还讨论了 RDB 和 AOF 之间的区别，以及如何将这两种方式结合起来使用。

第7章，配置高可用和集群（Cluster），主要涉及 Redis 的高可用相关架构。本章演示了如何配置 Redis Sentinel 和 Redis Cluster，并通过几个实验对 Redis Sentinel 和 Redis Cluster 的工作原理进行了解释。

第8章，生产环境部署，讨论了在生产环境中部署 Redis 时所要注意事项。本章首先讨论了操作系统、网络和安全方面的考虑；之后，涉及了配置调整和日志两个主题，也对 LRU 策略进行了讨论。最后，本章还讨论了 Redis 的性能/压力测试。

第9章，管理 Redis，主要涉及各种 Redis 的管理任务，包括更新服务器配置、使用 redis-cli、备份和恢复数据、管理内存使用、管理客户端和数据迁移等。

第10章，故障诊断，主要涉及几个有关排除 Redis 故障的实例。本章涵盖了使用慢日志来定位慢查询的例子，还演示了排除延迟、内存和进程崩溃等常见故障的案例。

第11章，通过模块扩展 Redis，讨论了如何使用 Redis 模块来扩展 Redis 的功能。本章讲解了 Redis 模块的工作原理，并演示了如何使用 Redis 模块 SDK 来构建 Redis 模块。

第12章，Redis 生态环境，讲解了 Redis 的第三方组件，还简要地介绍了几个流行的工具、客户端和代理。

附录 A，Windows 环境搭建，介绍了如何在 Windows 环境中运行 Redis。

0.3 如何更好地使用本书

本书中所有的例子都是在 Redis 4.x 上运行的。Linux 环境是首选，但也支持 macOS X。如果读者使用的是 Windows 操作系统，那么建议在 VirtualBox 或 VMware 中安装和运行一个 Linux 操作系统。此外，要运行代码示例还需要 JDK 1.8+ 及 Python 2.7+ 或 Python 3.4+。

0.3.1 下载示例代码

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/34081>



本书的示例代码包也同样托管在 GitHub 上，链接为 <https://github.com/PacktPublishing/Redis-4.x-Cookbook>。如果示例代码在本书出版后还有更新，那么将会更新到上述链接对应的 GitHub 库中。

在 <https://github.com/PacktPublishing> 上还有其他的代码及视频等资料。请读者自行查看！

0.3.2 下载彩色配图

本书还提供了一个 PDF 文件，该文件中包括了本书中所用截屏/图表的彩色图像。读者可以从以下链接下载：http://www.packtpub.com/sites/default/files/downloads/Redis4xCookbook_ColorImages.pdf。

0.3.3 惯例

本书中有一些行文的惯例。

• **文本中的代码：**表示文本中的代码、数据库表名、文件夹名、文件名、文件扩展名、路径名、占位 URL 和用户输入。例如：“打开一个终端并使用 `redis-cli` 连接到 Redis”。

一段代码形如：

```
for i in `seq 10`  
do  
nohup node generator.js hash 1000000 session:${i} &  
done
```

所有的命令行输入或输出都遵循如下的格式：

```
127.0.0.1:6379> SETBIT "users_tried_reservation" 100 1  
(integer) 0
```

粗体：表示一个新术语、一个重要的词或读者在屏幕上所看到的单词。例如，菜单或对话框中的单词。举一个例子：“点击 Import Project from Sources,然后在 coding 目录中选择 redis-4.0.1 子目录”。

0.4 小节

在本书中，读者会发现几个经常出现的标题（准备工作、操作步骤、工作原理、更多细节和相关内容）。

为了清晰地理解每一个实战案例，请按照如下的方式阅读每一小节。

0.4.1 准备工作

本节主要包括相应实战案例的主要内容，并描述了相应实战案例所需的软件和预先配置。

0.4.2 操作步骤

本节包含了完成相应实战案例所需的步骤。

0.4.3 工作原理

本节通常包括对上一节所发生的事情的详细解释。

0.4.4 更多细节

本节包含关于实战案例的额外信息，以便加深读者对实战案例的了解。

0.4.5 相关内容

本节提供了关于实战案例的其他有用信息的有用链接。

非卖品!!! 严禁(售卖和上传互联网平台)!!!
仅供对书籍质量进行鉴定甄别! 为是否购买正版实体书提供依据!!

译者序

2013 年,我在刚刚参加工作之初由于接触 ELK 而第一次听说了 Redis。当时,初次接触 Redis 是在 Logstash 中将其用作 Shipper 与 Indexer 之间的 Broker (队列缓冲,现在通常已经换为 Kafka 等具有更强消息堆积能力的消息中间件)。2013 年,随着内存成本的下降、内存数据库及分布式计算基础设施等的逐渐完善,实现了计算速度、计算能力和计算规模等的量变转向质变,成为了真正意义上的大数据元年。我是一名日常主要与 DB2/Oracle/MySQL 等关系型数据库打交道的传统银行科技从业人员,而 Redis 作为大数据基础设施的重要组成部分之一,为我打开了非关系型数据库的大门,使我意识到除了关系型数据库外还有键值存储、文档型数据等所谓 NoSQL 数据库的存在,也更加让我意识到,在能够真正服务于实际业务场景的系统架构中,往往需要上述多种异构数据库的有机结合。

作为一名 J2EE 开发工程师,我在工作中除了面向诸如 TOP N、全局缓存、全局会话、分布式锁、集合运算、基数计数、基于内存的对账、不严格可靠的消息队列等 Redis 常见的应用场景外,还需要使用 Redis 配合 Lua 脚本实现很多诸如“对性能和精确性均有严格要求的分布式 LRU (最近最少使用)算法”等基于传统关系型数据库难以甚至根本无法完成的需求。Redis 丰富的数据结构、极高的性能、原子性的操作等特性使其能够弥补关系型数据库的诸多不足;同时,Redis 所支持的高可用架构和水平扩展的集群特性等也为其用于真正的生产场景提供了强有力的支持。事实上,诸如新浪微博等具有上千个 Redis 实例、承载着天量 TPS 的 Redis 后端存储服务并不少见;Redis 在大量的一线互联网公司和其他领域的科技公司中均扮演着举足轻重的角色。

本书的原作者,两位来自不同领域的优秀一线工程师,黄鹏程和王左非,作为两个中国人,默默地用英文完成了本书的写作,并在国外知名的出版社 Packt 出版发行——我想,这本身就是一件非常值得称道的事。

纵观目前有关 Redis 的有限几本在售书籍,面向的主要还是 Redis 3.x 以下的陈旧版本,针对的往往也只是与 Redis 相关的某个方面,或偏重底层的设计与实现,或偏重运维实践,几乎没有一本书从 Redis 基础本身进行系统性的介绍,往往在读完一本书后仍然需要配合其他书籍或搭配文档才能用于实际工作;网络上的各类中文资料、教程、手册也大多是碎片化的,甚至有很多漏洞百出的劣质付费课程充斥其中,不利于初学者的学习。

本书针对的是 Redis 4.x 最新版本,它在写作上没有按照传统中式书籍顺序式照本宣科的写法,而是采用了当下西方更适用于技术类书籍的案例式组织方法。在翻译本书的过程中,透过作者精心设计的 80 余个实战案例,我能够深刻地感受到作者在通过这些案例让读者融入真正的业务场景所做的努力,也能深切地感受到作者在设计每一个案例或实验时的良苦用心。

通过这些精心规划的案例,读者不仅可以由浅入深地学习有关 Redis 的几乎所有知识,还可以

将案例中所讲解的内容直接用于包括设计、开发和运维等在内的各类生产实践。本书中的每一个案例，每一个案例中所涉及的各种知识、命令和工具等，均来自于作者从事一线企业级应用的总结；本书中总结的各类参数配置和故障诊断的案例等，也均来自作者的真实企业级运维工作经验。我相信，这样一本针对 Redis 最新版本的系统性书籍一定能够为诸多领域的学生、爱好者和从业人员提供帮助。

这是我翻译的第二本书，在翻译本书的过程中，我依然力图以“信、达、雅”的原则要求自己，从一名计算机行业一线从业者的角度，在尽可能正确地理解了原著英文意思后，用尽可能专业的语言进行表述，避免出现读者“感觉还不如直接去看英本原版”的情况。非常值得一提的是，与国内市面上其他很多译著不同的是，本书的两位作者也在繁重的日常工作中挤出了大量时间，对本书的初译稿进行了极为认真和深入的审阅、校对和修改，在本书完成的过程中付出了大量的心血和贡献。但是，受精力和能力所限，相信在译文中仍然会有诸多不妥、失误甚至错误出现，如果读者有任何意见或建议，可以直接通过我的邮箱（mlkui@163.com）或微信（微信号：MlkuiFly）或 QQ 读者交流群（QQ 群号：701534256）联系我，我虚心接受一切批评和指正。

最后，我要：

- 感谢本书两位原作者的信任和认可，给我提供了有幸翻译本书的机会；感谢他们放弃大量休息时间参与本书翻译的全过程，并在本书译稿校对等方面付出了大量心血；
- 感谢我的父母、妻子及亲人们多年来给予的无限关心、支持和陪伴，他们是我今天幸福生活的缔造者和组成者，也是我奋斗的根本动力和首要原因；
- 感谢我不便在此一一列举的领导和同事们，感谢他们一直以来在工作和生活中给予的无限支持、认可、包容和指点；
- 感谢中学、本科及研究生求学生涯中我不便在此一一列举的朋友、校友、同学、学长、老师、导师和团队，感谢他们多年以来给予的陪伴、分享、认可和信任，也祝愿我们在未来携手共创辉煌；
- 感谢电子工业出版社、电子工业出版社业界顶级品牌博文视点、博文视点孙学瑛策划编辑的认可 and 信任，感谢他们在本书引进并最终出版发行全过程的卓越眼光、专业能力、专业态度和极高的工作效率。

梅隆魁

2018 年 4 月于北京

目录

第 1 章 开始使用 Redis	1
1.1 本章概要	1
1.2 下载和安装 Redis	2
1.3 启动和停止 Redis	5
1.4 使用 redis-cli 连接到 Redis	8
1.5 获取服务器信息	10
1.6 理解 Redis 事件模型	13
1.7 理解 Redis 通信协议	22
第 2 章 数据类型	25
2.1 本章概要	25
2.2 使用字符串（string）类型	26
2.3 使用列表（list）类型	29
2.4 使用哈希（hash）类型	33
2.5 使用集合（set）类型	37
2.6 使用有序集合（sorted set）类型	40
2.7 使用 HyperLogLog 类型	43
2.8 使用 Geo 类型	46
2.9 键管理	48
第 3 章 数据特性	52
3.1 本章概要	52
3.2 使用位图（bitmap）	53
3.3 设置键的过期时间	55
3.4 使用 SORT 命令	57
3.5 使用管道（pipeline）	60
3.6 理解 Redis 事务（transaction）	62

3.7	使用发布订阅（PubSub）	65
3.8	使用 Lua 脚本	67
3.9	调试 Lua 脚本	70
第 4 章	使用 Redis 进行开发	75
4.1	本章概要	75
4.2	Redis 常见应用场景	76
4.3	使用正确的数据类型	78
4.4	使用正确的 API	87
4.5	使用 Java 连接到 Redis	92
4.6	使用 Python 连接到 Redis	98
4.7	使用 Spring Data 连接到 Redis	102
4.8	使用 Redis 编写 MapReduce 作业	107
4.9	使用 Redis 编写 Spark 作业	113
第 5 章	复制	118
5.1	本章概要	118
5.2	配置 Redis 的复制机制	118
5.3	复制机制的调优	125
5.4	复制机制的故障诊断	131
第 6 章	持久化	140
6.1	本章概要	140
6.2	使用 RDB	141
6.3	探究 RDB 文件	145
6.4	使用 AOF	149
6.5	探究 AOF 文件	151
6.6	RDB 和 AOF 的结合使用	155

第 7 章 配置高可用和集群	160
7.1 本章概要	160
7.2 配置 Sentinel	161
7.3 测试 Sentinel	166
7.4 管理 Sentinel	174
7.5 配置 Redis Cluster	178
7.6 测试 Redis Cluster	187
7.7 管理 Redis Cluster	195
第 8 章 生产环境部署	202
8.1 本章概要	202
8.2 在 Linux 上部署 Redis	202
8.3 Redis 安全相关设置	207
8.4 配置客户端连接选项	210
8.5 配置内存策略	212
8.6 基准测试	215
8.7 日志	218
第 9 章 管理 Redis	222
9.1 本章概要	222
9.2 管理 Redis 服务器配置	223
9.3 使用 bin/redis-cli 操作 Redis	225
9.4 备份和恢复	228
9.5 监控内存使用情况	230
9.6 管理客户端	237
9.7 数据迁移	241
第 10 章 Redis 的故障诊断	248
10.1 本章概要	248
10.2 Redis 的健康检查	248
10.3 使用 SLOWLOG 识别慢查询	252

10.4 延迟问题的故障诊断	254
10.5 内存问题的故障诊断	262
10.6 崩溃问题的故障诊断	270
第 11 章 使用 Redis 模块扩展 Redis	276
11.1 本章概要	276
11.2 加载 Redis 模块	277
11.3 编写 Redis 模块	279
第 12 章 Redis 生态系统	286
12.1 本章概要	286
12.2 Redisson 客户端	287
12.3 Twemproxy	288
12.4 Codis——一个基于代理的高性能 Redis 集群解决方案	289
12.5 CacheCloud 管理系统	291
12.6 Pika——一个与 Redis 兼容的 NoSQL 数据库	292
附录 A Windows 环境搭建	294

第 1 章

开始使用 Redis

在本章中，我们将学习下列案例：

- 下载和安装 Redis。
- 启动和停止 Redis。
- 使用 redis-cli 连接到 Redis。
- 获取服务器信息。
- 理解 Redis 事件驱动模型。
- 理解 Redis 通信协议。

1.1 本章概要

Redis 是一个非常流行的基于内存的轻量级键值数据库（key-value database）。严格地说，按照 Redis 重要贡献者之一 **Matt Stancliff**（@mattsta）的说法，与其把 Redis 称为一种数据库，不如说 Redis 是一种数据结构服务器更为恰当（<https://matt.sh/thinking-in-redis-part-one>）。Redis 的作者 **Salvatore Sanfilippo**（@antirez）起初将其叫作 Redis，代表 **REmote DIctionary Server**。这是因为 Redis 原生地在内存中实现了多种类型的数据结构，并提供了操作这些数据结构的多种 API。更重要的是，作为一个需要长期运行的数据存储服务，Redis 还提供了高性能命令处理、高可靠性/扩展性的架构及数据持久化等特性。

随着高并发、低延迟系统的发展，Redis 的使用正在变得越来越广泛。自 2017 年起，Redis 就在 DB-Engine 排行榜（DB-Engine complete ranking, <https://db-engines.com/en/ranking>）中排到了前十。在此之前，Redis 还一直占据着 DB-Engine 键值存储分类榜单的第一名相当长一段时间。

本章的目标是带领读者快速搭建一个简单的 Redis 实例，并学习诸如启动、连接和停止一个 Redis 服务器等常用操作。另外，还介绍了如何从一个 Redis 服务器中获取基本信息。本章最后两小节对 Redis 的事件模型及通信协议进行了详细的讨论。

1.2 下载和安装 Redis

Redis 在 GitHub 上有一个活跃的社区。在过去几年间已经合并了大量的 Pull Request，而作者 **Antirez** 也一直在 GitHub 上及时地回复问题。因此，Redis 的发布周期很短。从曾被广泛使用的早期版本 2.6/2.8 到 3.0/3.2，再到最新的 4.x，每次的发布都包含了一些重要的新功能、性能提升和缺陷修复。因此，如果可能的话，“使用最新版的 Redis”本身就是最佳实践之一。在本书中，我们采用的是写作时最新版的 Redis 4.0.1。

Redis 是一个完全用 C 语言编写的开源软件，因而我们可以自行编译并安装。大部分操作系统都在其软件仓库中预置了 Redis 的二进制可执行文件，但这些软件仓库中的 Redis 版本通常会有点陈旧。

1.2.1 准备工作

读者可以从 <https://redis.io/download> 中找到下载链接和基本的安装步骤。如果读者想自己在 Linux/Unix/macOS 系统上从源码编译构建 Redis，那么还需要在你的环境中安装 gcc 编译器和 libc。如果需要通过操作系统的预发布软件仓库进行安装，那么只需要连接互联网并正确配置好软件仓库即可。

1.2.2 操作步骤

我们将在 Ubuntu 16.04.2 LTS (Xenial Xerus) 中演示 Redis 的编译和安装。下载及构建的步骤如下。

1. 安装编译工具：

```
$ sudo apt-get install build-essential
```

2. 为 Redis 创建目录并切换到所创建的目录中：

```
$ mkdir /redis  
$ cd /redis
```

3. 下载 Redis：

```
$ wget http://download.redis.io/releases/redis-4.0.1.tar.gz
```

4. 解压下载到的 Redis 源码并切换到对应的目录下：

```
$ tar zxvf redis-4.0.1.tar.gz  
$ cd redis-4.0.1
```

5. 为 Redis 的配置文件创建目录并把默认配置文件复制进去：

```
$ mkdir /redis/conf  
$ cp redis.conf /redis/conf/
```

6. 编译依赖项：

```
$ cd deps  
$ make hiredis lua jemalloc linenoise  
$ cd ..
```

注意

由于不同操作系统及安装在操作系统中的库之间存在差异，此前提到的步骤可能会出现缺少某些依赖的错误。例如，读者可能会碰到如下的错误消息：`zmalloc.h:50:31: fatal error: jemalloc/jemalloc.h: No such file or directory`。如果没有依赖项相关的报错，则这一步并非是必需的。

7. 编译 Redis：

```
$ make
```

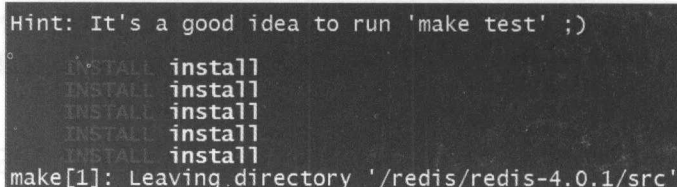
如果编译顺利，将看到如下的提示，代表已经成功地完成了编译：

```
It's a good idea to run 'make test' ;)  
make[1]: Leaving directory '/redis/redis-4.0.1/src'
```

8. 安装 Redis：

```
$ make PREFIX=/redis install
```

出现图1.1代表已经成功地完成了安装。



```
Hint: It's a good idea to run 'make test' ;)
INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install
INSTALL install
make[1]: Leaving directory '/redis/redis-4.0.1/src'
```

图 1.1 安装成功

9. 进入 `/redis` 目录并验证生成了 Redis 的二进制可执行文件：

```
$ ls /redis/bin
redis-benchmark redis-check-aof redis-check-rdb redis-cli redis-sentinel
redis-server
```

恭喜！这样就完成 Redis 的编译和安装了。

与编译和安装相比，在 Ubuntu 中使用 `apt-get` 安装 Redis 要容易得多。具体步骤是：

1. 首先，更新软件仓库的索引：

```
$ sudo apt-get update
```

2. 然后安装：

```
$ sudo apt-get install redis-server
```

3. 安装完成后，可以使用如下命令来验证 Redis 是否已经在您的环境中被正确地安装了：

```
$ which redis-server
```

1.2.3 工作原理

当涉及 Redis 版本的选择时，请记住 Redis 遵循如下的标准版本编号实践，即 *major.minor.patch*（主版本号.次版本号.补丁版本号）的层级形式。偶数的主版本号代表稳定版，奇数的主版本号表示不稳定版（虽然 Redis 也有少数几个版本使用了奇数的主版本号）。

编译安装和通过软件仓库安装 Redis 的不同之处在于，前者可以在编译时添加优化或调试选项，还能够灵活地指定安装位置。

安装完成后，bin 目录中会有一些可执行文件。关于它们的介绍和备注如表 1.1 所示。

表 1.1 bin 目录中的可执行文件

文件名	描述	备注
redis-server	Redis 服务端	
redis-sentinel	Redis Sentinel	redis-server 的软链接
redis-cli	Redis 命令行工具	
redis-check-rdb	Redis RDB 检查工具	
redis-check-aof	Redis Append Only Files （AOF）检查工具	
redis-benchmark	Redis 基准/性能测试工具	

1.2.4 更多细节

对于 Windows 操作系统来说，微软开源技术小组（Microsoft Open Technologies group）曾经维护了一个 Windows 的 Redis 发行版，读者可以从<https://github.com/MicrosoftArchive/redis/releases>获取该版本。

读者只需下载 .msi 可执行文件，然后双击打开，并按照默认配置安装即可。

对于 macOS 操作系统而言，与 Linux 操作系统下的编译安装没有什么太大区别。读者还可以通过使用 `brew install redis` 命令来安装 Redis。

1.2.5 相关内容

- 有关不同编译选项对 Redis 性能的影响，请参考 Matt Stancliff 在使用不同编译选项时对不同版本 Redis 性能影响的评估：<https://matt.sh/redis-benchmark-compilers>。
- 出于安全方面的考虑，Redis 应该运行在非 root 权限下。本书第 8 章生产环境部署中的 Redis 安全相关设置一节会进行详细的讨论。
- 读者可以参阅<https://github.com/antirez/redis>获取更多的信息。

1.3 启动和停止 Redis

在使用 Redis 前，我们必须正确地启动 Redis 服务。同样，在某些情况下，我们又不得不停止 Redis 服务。本案例将向读者展示如何启动和停止 Redis 服务端。

1.3.1 准备工作

我们需要按照本章下载和安装 *Redis* 的案例中所描述的步骤完成 *Redis* 服务器的安装。

1.3.2 操作步骤

启动和停止 *Redis* 服务端的步骤如下。

1. 我们可以使用默认配置来启动一个 *Redis* 实例：

```
$ bin/redis-server
```

服务端此时应该会启动，如图1.2所示。

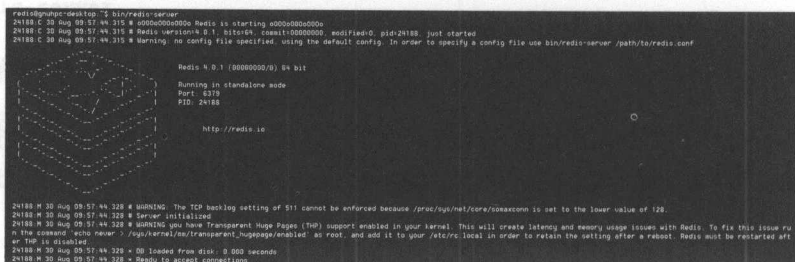


图 1.2 启动服务端

2. 要在启动 *Redis* 服务端时指定配置文件，例如欲使用我们从源码包中拷贝过来的配置文件时，可以使用如下的命令：

```
$ bin/redis-server conf/redis.conf
```

3. 如果是从操作系统的软件仓库中安装的 *Redis*，那么可以使用 *init.d* 脚本启动 *Redis*：

```
$ /etc/init.d/redis-server start
```

4. 如果要以 *redis-server* 守护进程的方式在后台启动 *Redis*，那么可以编辑配置文件并将 *daemonize* 参数设为 *yes* 并使用该配置文件启动：

```

$ vim conf/redis.conf
daemonize yes
$ bin/redis-server conf/redis.conf
  
```

图1.3中的提示信息 *Configuration loaded* 说明配置文件已经生效了：

```
redis@gnuhpc-desktop:~$ bin/redis-server conf/redis.conf
23527:C 30 Aug 09:10:58.089 # 000000000000 Redis is starting 000000000000
23527:C 30 Aug 09:10:58.089 # Redis version=4.0.1, bits=64, commit=00000000, modified=0, pid=23527, just started
23527:C 30 Aug 09:10:58.089 # Configuration loaded
```

图 1.3 配置文件生效

5. 相应地，我们可以使用 Ctrl+C（如果 Redis 是以前台模式启动的），或 Kill + PID（如果 Redis 是以后台模式启动的）来停止 Redis 服务：

```
$ kill `pidof redis-server`
```

6. 更加优雅和推荐的停止 Redis 的方式是通过 redis-cli 调用 shutdown 命令：

```
$ cd /redis
$ bin/redis-cli shutdown
```

7. 如果 Redis 是从软件仓库中安装的话，那么还可以通过 init.d 脚本关闭：

```
$ /etc/init.d/redis-server stop
```

1.3.3 工作原理

Redis 中的术语实例代表一个 redis-server 进程。同一台主机上可以运行多个 Redis 实例，只要这些实例使用不同的配置即可，比如绑定到不同的端口、使用不同的路径保存数据持久化相关的文件，或采用不同的日志路径等。

启动和停止 Redis 实例是基本操作。对于启动 Redis 而言没有太多需要注意的；但对于一个数据服务来说，停止 Redis 服务却尤为需要注意，因为作为一种数据存储服务，学会如何优雅地停止 Redis 服务端以保持数据的一致性，是非常重要的。

之所以强烈建议使用 shutdown 命令停止 Redis 服务的原因在于，如果我们关心数据一致性且配置了数据持久化来将内存中的数据保存到磁盘中（Redis 数据持久化将在第 6 章持久化中讨论），那么 shutdown 命令发出后，除了终结进程外，还会执行一系列的其他操作。

首先，redis-server 会停止响应客户端的连接；然后，如果启用了持久化，则会执行数据持久化操作。之后，如果 .pid 文件和 socket 套接字文件描述符存在的话，则对其进行清理，并最终退出进程。通过这种策略，Redis 会尽可能地防止数据丢失。相反，如果粗暴地使用 kill 命令来终止 redis-server 进程，那么由于在服务端关闭之前数据可能尚未被持久化而导致数据丢失。

应该注意的是，使用 kill 命令或其他进程管理工具向 Redis 进程发送 SIGTERM (15) 信号基本上等同于使用 shutdown 命令优雅地停止 redis-server。

1.3.4 更多细节

在启动 Redis 时,可以直接将配置参数添加到 `redis-server` 的命令行参数中,这对于在单台主机上部署多个实例的情况非常有用。当单台主机上存在多个实例时,我们可以把通用的配置参数保存在一个配置文件中;同时,在启动 Redis 时通过命令行参数传递每个实例所独有的配置参数。过这种方式,可以降低维护多个配置文件的成本,并且可以通过 `ps` 或其他系统命令轻松地区分各个实例。

此外,我们还可以使用诸如 `systemd`、`supervisord` 或 `Monit` 等进程管理工具来管理 Redis 实例。这可以防止在单台主机上部署多个实例时混淆不同的实例。我们唯一需要注意的是前面提到的启动时命令行参数和退出信号的处理机制。

1.3.5 相关内容

- 请参阅<https://redis.io/topics/signals>来学习更多关于 Redis 如何处理不同类型信号的逻辑。特别注意,这些信号处理机制之间细微但重要的区别。另外,请参阅<https://redis.io/commands/shutdown>来了解有关优雅地关闭 Redis 实例的更多细节。
- 对于使用进程管理工具来控制 Redis 启停的问题,<https://git.io/v5chR>中给出了一个使用 `systemd` 来控制 Redis 的配置示例。
- 此外,读者还可以参考第 6 章 持久化来学习 Redis 持久化相关的知识。

1.4 使用 redis-cli 连接到 Redis

在 Redis 相关的开发和运维工作中,bin 目录中的 `redis-cli` 是最常用的工具。本节将简要描述它的用法,以便读者能够了解如何通过 `redis-cli` 连接到 Redis 和使用 Redis。

1.4.1 准备工作

我们需要按照本章启动和停止 Redis 一节中所描述的步骤配置并启动一个 Redis 服务器。

1.4.2 操作步骤

使用 `redis-cli` 连接到 Redis 服务器的步骤如下。

1. 打开一个终端,并通过 `redis-cli` 连接到 Redis:

```
$ bin/redis-cli
127.0.0.1:6379>
```


命令行提示符前的提示符是 IP 地址:Redis 监听端口号的形式，这表示 `redis-cli` 已经成功地连接到了该 Redis 实例。

2. 我们可以发送一些简单的命令进行测试，其他更多的数据类型和特性将在接下来的章节中进行讨论。

(1) 设置两个字符串键值对 `foo value1` 和 `bar value2`:

```
127.0.0.1:6379> set foo value1
OK
127.0.0.1:6379> set bar value2
OK
```

(2) 获取刚刚设置的值:

```
127.0.0.1:6379> get foo
"value1"
127.0.0.1:6379> get bar
"value2"
```

(3) 通过发送 `shutdown` 命令来停止 Redis 实例:

```
$ bin/redis-cli
127.0.0.1:6379> shutdown
not connected>
```

(4) 在 Redis 服务器关闭后，命令行提示符会变为 `not connected`。然后，我们退出 `redis-cli`，并再次使用 `redis-cli` 进行连接。这时，会看到如下的错误信息:

```
not connected>quit
$ bin/redis-cli
Could not connect to Redis at 127.0.0.1:6379: Connection refused
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected>
```

1.4.3 工作原理

在默认情况下，`redis-cli` 会连接到 `127.0.0.1:6379`（默认端口）上运行的 Redis 实例。我们也可以使用 `-h` 选项指定要连接到的主机名/IP 地址，只需确保 `redis-cli` 和 Redis 服务器端之间的网络连接没有问题即可。

如果 Redis 服务器没有运行在默认的 `6379` 端口上的话，可以使用 `redis-cli` 的 `-p` 选项指定端口号。如果同一主机上运行了绑定不同端口号的多个 Redis 实例，这个选项也很有用。

另外，如果 Redis 实例启动了连接密码，那么可以使用 `-a` 选项在连接到 Redis 时指定密码。

此外，如果 Redis 实例启用了 Unix 套接字文件，那么只需使用 `-s` 选项来指定 Unix 套接字文件即可直接连接到 Redis 服务器了。

1.4.4 更多细节

在让应用程序直接操作 Redis 前，我们通常需要进行一些数据原型验证工作。`redis-cli` 对此项工作而言是一个非常有用的工具，它提供了一个交互式命令行界面，可供我们快速验证数据结构的设计。在 Redis 服务器的日常维护中，`redis-cli` 还提供了一组命令，包括获取各项指标、管理系统状态及执行参数配置等。

1.4.5 相关内容

- 请参阅第 9 章管理 Redis 中有关如何使用 `redis-cli` 管理一个 Redis 实例的详细讨论。

1.5 获取服务器信息

我们可以通过 `redis-cli` 的 `INFO` 命令来获得一个 Redis 实例最全面和重要的信息。在本节中，我们将学习如何使用 `INFO` 命令来获取这些基本的统计信息。

1.5.1 准备工作

我们需要按照本章启动和停止 Redis 一节中所描述的步骤配置并启动一个 Redis 服务器。

1.5.2 操作步骤

接下来，让我们学习如何获取一个 Redis 实例的服务器信息。

1. 连接到一个 Redis 实例，然后执行 `INFO` 命令：

```
$ bin/redis-cli
127.0.0.1:6379> INFO
```

结果形如：

```
# Server
redis_version:4.0.1
...
# Clients
connected_clients:1
...
# Memory
used_memory:828352
used_memory_human:808.94K
used_memory_rss:9420800
used_memory_rss_human:8.98M
...
# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1504223311
...
# Stats
total_connections_received:1
total_commands_processed:1
instantaneous_ops_per_sec:0
...
# Replication
role:master
connected_slaves:0
...
# CPU
used_cpu_sys:0.01
used_cpu_user:0.00
...
# Cluster
```

```
cluster_enabled:0
```

2. 我们可以通过增加一个可选的 <section> 参数来指定要获取哪一部分信息。

例如，我们可以通过 redis-cli 发送 INFO memory 来获得内存相关的指标，如图1.4所示。

```
# Memory
used_memory:827328
used_memory_human:807.94K
used_memory_rss:9420800
used_memory_rss_human:8.98M
used_memory_peak:828384
used_memory_peak_human:808.97K
used_memory_peak_perc:99.87%
used_memory_overhead:815118
used_memory_startup:765488
used_memory_dataset:12210
used_memory_dataset_perc:19.74%
total_system_memory:67467218944
total_system_memory_human:62.83G
used_memory_lua:37888
used_memory_lua_human:37.00K
maxmemory:0
maxmemory_human:0B
maxmemory_policy:noeviction
mem_fragmentation_ratio:11.39
mem_allocator:jemalloc-4.0.3
active_defrag_running:0
lazyfree_pending_objects:0
```

图 1.4 内存相关的指标

3. 另一种从一个 Redis 实例中获取状态信息的方式是直接 在 shell 命令行中使用 redis-cli INFO 命令。使用这种方式，可以非常方便地将命令的输出通过管道重定向到一个脚本中来进行指标分析或性能监控。

1.5.3 工作原理

INFO 命令会输出当前所连接到的 Redis 实例的所有指标，每个指标的格式形如 metric-name: metric-value，这种格式可以在后续很容易地进行解析。

表 1.2 总结了 INFO 命令所返回信息的全部段落。

表 1.2 INFO 命令的返回信息

段落名称	描述
Server	关于 Redis 服务器的基本信息
Clients	客户端连接的状态和指标
Memory	大致的内存消耗指标
Persistence	数据持久化相关的状态和指标
Stats	总体统计数据

表 1.2 INFO 命令的返回信息

Replication	主从复制相关的状态和指标
CPU	CPU 使用情况
Cluster	Redis Cluster 的状态
Keyspace	数据库相关的统计数据

1.5.4 更多细节

构建 Redis 监控应用的常见实践之一，就是通过定期地使用 INFO 命令来获取信息。

1.5.5 相关内容

- 请参阅第 10 章 Redis 故障诊断中的健康检查、延迟问题的故障诊断和内存问题的故障诊断等小节来了解有关 INFO 命令在 Redis 运维和故障诊断中的更多详细使用方法。
- 读者还可以进一步参阅<https://redis.io/commands/INFO>，该链接中列出了 INFO 命令所返回的所有指标的含义。

1.6 理解 Redis 事件模型

Redis 以其高性能而闻名，它最大程度地利用了单线程、非阻塞的 I/O 模型来快速地处理请求。因此，理解 Redis 的事件模型是非常必要的。为了让读者理解此模型，本案例首先展示了一个基于 Redis 异步事件库（ae 库）构建的 echo server 示例程序。然后，通过分析 Redis 源代码的核心片段，对 Redis 事件处理模型所涉及的重点进行了讲解。

本案例中涉及很多 C 语言的编程实践。因此，如果读者对 C 语言不熟悉，那么可以根据自己的意愿跳过本案例。跳过本节并不会影响对后续章节的理解。

1.6.1 准备工作

本案例涉及源码的构建和调试。因此，读者需要首先完成本章中下载和安装 Redis 的案例。为了更好地进行说明，我们还需要一个支持 C 语言编程的集成开发环境（IDE）。这里，我们所使用的集成开发环境是运行在 Ubuntu Desktop 16.04.3 LTS 上的 CLion。虽然 CLion 并不是免费的，但是 30 天的免费试用期对我们来说已经足够了。

我们还需要准备好 C 语言的编译器和开发环境。在 Ubuntu 中，可以通过以下的命令来安装相关的软件包：


```
$ sudo get update && apt-get install build-essential
```

安装完毕后，我们应确保 CMake 的版本为 3.5 或更高版本：

```
$ cmake --version  
cmake version 3.5.1  
  
CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

1.6.2 操作步骤

接下来，让我们按照以下的步骤来学习 Redis 事件模型。

1. 解压 Redis 的源代码包，然后手工而不使用 CMake 来构建一些所需的依赖：

```
~$ mkdir coding; cd coding  
~/coding$ tar xzvf redis-4.0.1.tar.gz  
~/coding$ cd redis-4.0.1/deps/  
~/coding/redis-4.0.1/deps$ make lua linenoise hiredis
```

图1.5表示依赖已经被成功构建了：



```
make[1]: Entering directory '/redis/coding/redis-4.0.1/deps/hiredis'  
cc -std=c99 -pedantic -c -O3 -fPIC -Wall -W -Wstrict-prototypes -Wwrite-strings -g -ggdb net.c  
cc -std=c99 -pedantic -c -O3 -fPIC -Wall -W -Wstrict-prototypes -Wwrite-strings -g -ggdb hiredis.c  
cc -std=c99 -pedantic -c -O3 -fPIC -Wall -W -Wstrict-prototypes -Wwrite-strings -g -ggdb sds.c  
cc -std=c99 -pedantic -c -O3 -fPIC -Wall -W -Wstrict-prototypes -Wwrite-strings -g -ggdb async.c  
cc -std=c99 -pedantic -c -O3 -fPIC -Wall -W -Wstrict-prototypes -Wwrite-strings -g -ggdb read.c  
ar rcs libhiredis.a net.o hiredis.o sds.o async.o read.o  
make[1]: Leaving directory '/redis/coding/redis-4.0.1/deps/hiredis'
```

图 1.5 配置文件生效

2. 下载 CLion 集成开发环境，然后将其解压到/redis/coding/文件夹中：

```
~/coding$ wget https://download.jetbrains.com/cpp/CLion-2017.2.2.tar.gz  
~/coding$ tar xzvf CLion-2017.2.2.tar.gz
```

3. 下载示例程序，用于 redis-server 的构建和调试。

4. 将其解压到 redis-4.0.1 目录：

```
~/coding$ tar xzvf echodemo.tar.gz -C redis-4.0.1/
```

确认 redis-4.0.1 文件夹中存在文件 CMakeLists.txt 和 echodemo，如图1.6所示。

```
redis@gnu-hpc-desktop:~/coding/redis-4.0.1$ ls
00-RELEASENOTES  CMakeFiles  COPYING  Makefile  deps  redis.conf  runtest-sentinel  tests
BUGS             CMakeLists.txt  INSTALL  README.md  echodemo  runtest  sentinel.conf  utils
CMakeCache.txt  CONTRIBUTING  MANIFESTO  cmake_install.cmake  redis.cbp  runtest-cluster  src
```

图 1.6 存在文件 CMakeLists.txt 和 echodemo

5. 登录 Ubuntu 桌面环境，并打开一个终端来启动 CLion：

```
~/coding/clion-2017.2.2$ bin/clion.sh
```

6. 在接受 CLion 的许可之后，我们需要确认 CMake 和调试器是可用的，如图1.7所示：

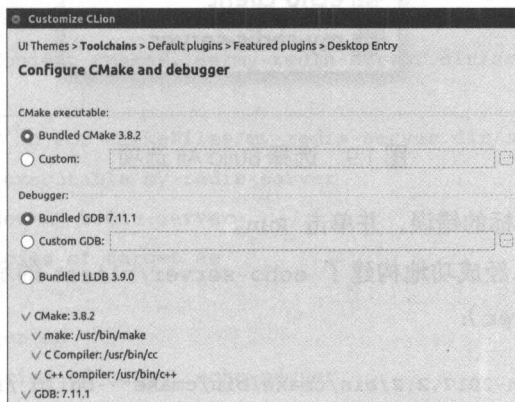


图 1.7 确认 CMake 和调试器是可用的

7. 使用默认选项进行下一步，直到出现图1.8所示的截图：

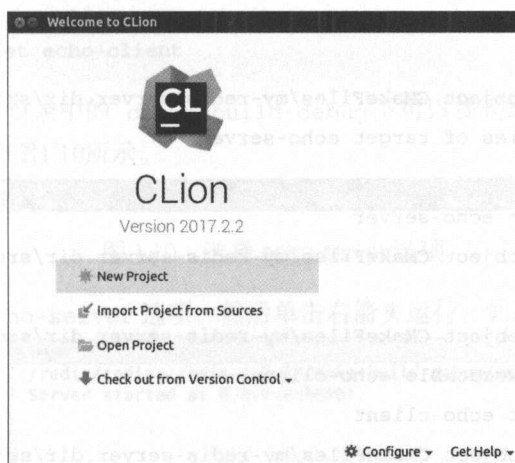


图 1.8 CLion 主界面

8. 单击 Import Project from Sources 并选择 coding 目录中的 redis-4.0.1 子文件夹。

9. 单击 OK 按钮，然后选择 Open Project 来打开一个 Redis 工程。
10. 选择右上角的 Build All 选项，然后单击 Run 按钮，如图1.9所示。

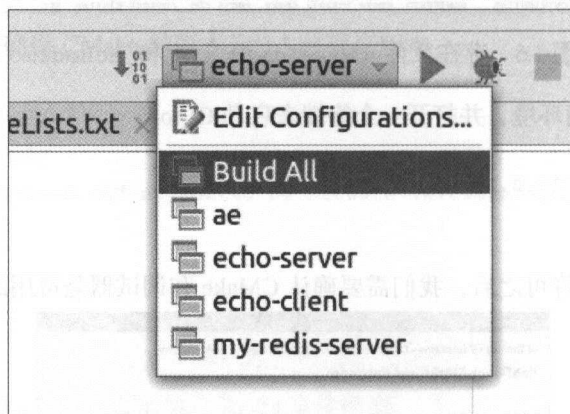


图 1.9 选择 Build All 选项

忽略没有指定可执行目标的错误，并单击 Run。

下面的日志表示我们已经成功地构建了 echo-server/client 的示例程序和 redis-server（本例中是 my-redis-server）：

```
/redis/coding/clion-2017.2.2/bin/cmake/bin/cmake --build /redis/coding/redis
-4.0.1/cmake-build-debug --target all -- -j 6
Scanning dependencies of target ae
Scanning dependencies of target my-redis-server
[ 1%] Building C object CMakeFiles/ae.dir/src/zmalloc.c.o
...
[ 17%] Building C object CMakeFiles/my-redis-server.dir/src/blocked.c.o
Scanning dependencies of target echo-server
...
[ 25%] Built target echo-server
[ 26%] Building C object CMakeFiles/my-redis-server.dir/src/config.c.o
...
[ 31%] Building C object CMakeFiles/my-redis-server.dir/src/defrag.c.o
[ 32%] Linking C executable echo-client
[ 32%] Built target echo-client
[ 98%] Building C object CMakeFiles/my-redis-server.dir/src/zmalloc.c.o
[100%] Linking C executable my-redis-server
[100%] Built target my-redis-server
```


11. 如果读者想直接使用命令行来编译示例程序，可以按以下的步骤进行。

```
/redis/coding/redis-4.0.1$ cmake
-- The C compiler identification is GNU 5.4.0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /redis/coding/redis-4.0.1
/redis/coding/redis-4.0.1$ make
Scanning dependencies of target my-redis-server
[ 1%] Building C object CMakeFiles/my-redis-server.dir/src/adlist.c.o
[ 2%] Building C object CMakeFiles/my-redis-server.dir/src/ae.c.o
...
[ 85%] Building C object CMakeFiles/my-redis-server.dir/src/zmalloc.c.o
[ 86%] Linking C executable my-redis-server
[ 86%] Built target my-redis-server
Scanning dependencies of target ae
...
[ 92%] Built target ae
Scanning dependencies of target echo-server
...
[ 96%] Built target echo-server
Scanning dependencies of target echo-client
...
[100%] Built target echo-client
```

12. 在 redis-4.0.1 目录中的 cmake-build-debug 下可以找到编译出的 echo-server、echo-client 和 my-redis-server，如图1.10所示。

```
redis@gnuhpc-desktop:~/coding/redis-4.0.1/cmake-build-debug$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake dump.rdb echo-server libae.a Makefile my-redis-server redis.cbp
```

图 1.10 选择 echo-server 选项

13. 选择右上角的 echo-server 选项，然后单击右箭头运行，如图1.11所示：

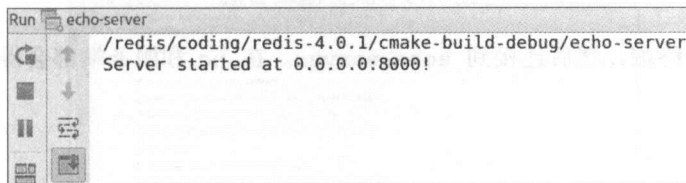


图 1.11 echo-server 的输出

14. 打开一个终端, 然后使用 nc (Netcat) 连接到 echo-server:

```
~/coding/redis-4.0.1/cmake-build-debug$ nc 127.0.0.1 8000
```

15. 如果服务端启动成功的话, 会输出提示 Hello Client!, 如图1.12所示:

```
gnuohpc@gnuohpc-desktop:~$ nc 127.0.0.1 8000  
Hello Client!
```

图 1.12 成功连接到服务端的输出

服务器上也会输出连接相关的日志, 如图1.13所示:

```
/redis/coding/redis-4.0.1/echo-server  
Server started at 0.0.0.0:8000!  
Accept new connection in acceptProc.  
Client info - ip 127.0.0.1 port 34998  
AnetNonBlock running successfully
```

图 1.13 服务器端输出的日志

16. 在 nc 中输入 Hello, please echo! 然后按回车将其发送出去。服务端会将其原样返回, 如图1.14所示。

```
gnuohpc@gnuohpc-desktop:~$ nc 127.0.0.1 8000  
Hello Client!  
Hello, please echo!  
Hello, please echo!
```

图 1.14 使用 nc 发送向服务端发送内容并被原样返回

服务端会在日志中输入接收到的数据, 然后将其发送回客户端, 如图1.15所示:

```
Reading Client data from 127.0.0.1:46728!  
Size of Data to be written:20, and the Data is: Hello, please echo!  
  
Sending Client data to 127.0.0.1:46728!  
Size of Sent Data:20, and the Data is: Hello, please echo!
```

图 1.15 服务器端日志

17. 打开另一个终端, 然后连接到 echo-server。在 nc 和服务端都会得到类似的结果, 如图1.16所示。

```

Accept new connection in acceptProc.
Client info - ip 127.0.0.1 port 35004
AnetNonBlock running successfully

Reading Client data from 127.0.0.1:48264!
Size of Data to be written:13, and the Data is: Hello again!

Sending Client data to 127.0.0.1:48264!
Size of Sent Data:13, and the Data is: Hello again!

```

图 1.16 另一个终端连接后的日志

18. 如果需要的话，我们可以调试 echo-server，如图1.17所示：

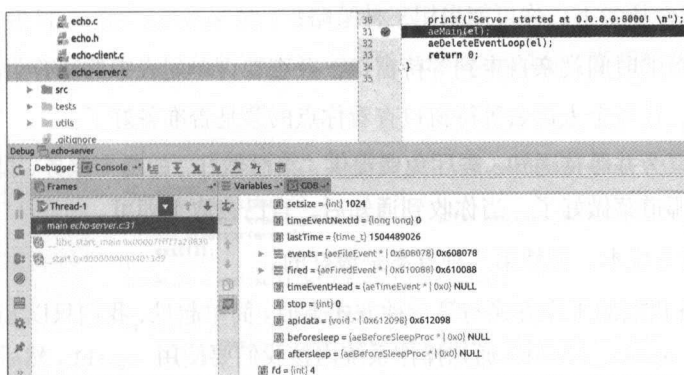


图 1.17 调试 echo-server

19. 在此基础上，我们可以用与 echo-server 示例差不多一样的方式构建和调试 redis-server（在本例中称为 my-redis-server），来深入研究源码。唯一需要修改的是选择 my-redis-server 的运行/调试配置，如图1.18所示：

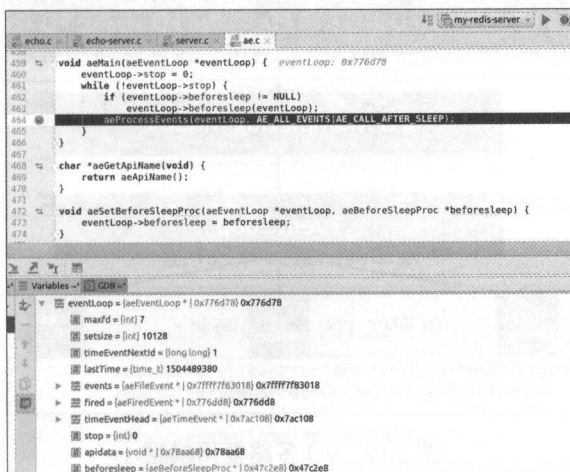


图 1.18 调试 my-redis-server

1.6.3 工作原理

正如此前所提到的，Redis 极大程度地获益于其单线程、非阻塞、多路复用的 I/O 模型。当然，在某些情况下，Redis 也会创建线程或子进程来执行某些任务。

Redis 包含了一个简单但功能强大的异步事件库，称为 `ae`。该库中封装了不同操作系统的 polling 机制（译者注：即非阻塞 I/O 相关的机制），如 `epoll`、`kqueue` 和 `select` 等。

那么操作系统的 polling 机制究竟是什么呢？我们可以使用一个真实的场景进行说明。试想：你在一家餐厅点了五道菜，你必须自己在一个等候窗口上取回你点的菜，由于你已经很饿了，因此你想尽快拿到菜。在这个场景下，你可能采用三种策略：

- 每隔一段很短的时间就亲自走到等待窗口，并查看订单列表中的所有菜品是否都准备好了。
- 雇佣五个人，让每个人都去等待窗口查看你点的菜是否准备好了。
- 仅仅是坐在桌旁并等待通知。餐厅免费提供“菜已做好”的通知服务，即会有服务员在菜做好后通知你哪道菜做好了。当你收到通知后，自己去取菜即可。

考虑到时间和精力成本，显然第三种方案是最好的。

操作系统 polling 机制的工作方式与第三种方式类似。简单起见，我们只以 Linux 中的 `epoll` 为例。首先，我们调用 `epoll_create` 通知操作系统内核我们要使用 `epoll`。然后，调用 `epoll_ctl` 把文件描述符（**FD**）和所关注的事件类型传给内核。之后，调用 `epoll_wait` 等待通过 `epoll_ctl` 所设置的文件描述符上发生所关注的事件。当文件描述符被更新时，内核会向应用程序发送通知。我们唯一需要做的就是为我们所这些关注的这些事件创建事件处理函数/回调函数。

上述多路复用的全过程如图 1.19 所示：

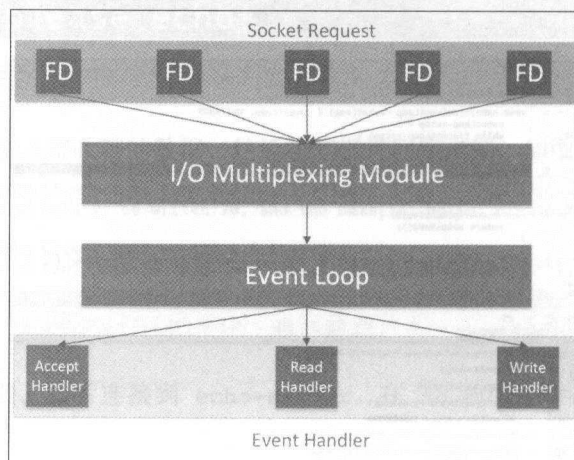


图 1.19 I/O 多路复用模型

Redis 的 `ae` 库基本上就是按照此前描述的过程来处理请求的。在 `echo-server` 的示例中，我们

首先调用 `aeCreateEventLoop` 创建了一个事件循环。然后通过 `anetTcpServer` 创建了一个 TCP 服务器，用于绑定端口和监听网络请求。之后，调用 `anetNonBlock` 将套接字 FD 设置非阻塞 I/O 模式。接下来，我们把套接字 FD 的“客户端连接”事件的回调函数设置为 `acceptProc`，该回调函数被在 `aeCreateEventLoop` 中创建的事件循环使用。当 TCP 连接建立后，服务器将触发 `acceptProc` 中的动作。在 `acceptProc` 中，我们使用 `anetTcpAccept` 接受连接请求，并将套接字 FD 的“有数据可供读取”事件的回调函数注册为 `readProc`。之后，当 `readProc` 被调用时，我们就可以读取到发给服务器的数据，并设置套接字 FD 的“有数据可供写入”事件的回调函数。随后，当事件循环接收到“有数据可供写入”事件时，就会触发 `writeProc` 将数据发送给套接字的客户端。

Redis 的工作方式与 `echo-server` 的工作方式基本相同。`server.c` 主函数中的 `initServer` 方法调用了 `aeCreateEventLoop`、`anetTcpServer` 和 `anetNonBlock` 来初始化服务器，如图 1.20 所示：

```

1812     createSharedObjects();
1813     adjustOpenFilesLimit();
1814     server.el = aeCreateEventLoop(server.maxclients+CONFIG_FDSET_INCR);
1815     if (server.el == NULL) {
1816         serverLog(LL_WARNING,
1817             "Failed creating the event loop. Error message: '%s'",
1818             strerror(errno));
1819         exit(1);
1820     }
1821     server.db = zmalloc(sizeof(redisDb)*server.dbnum);
1822
1823     /* Open the TCP listening socket for the user commands. */
1824     if (server.port != 0 &&
1825         listenToPort(server.port,server.ipfd,&server.ipfd_count) == C_ERR)
1826         exit(1);
1827
1828     /* Open the listening Unix domain socket. */
1829     if (server.unixsocket != NULL) {
1830         unlink(server.unixsocket); /* don't care if this fails */
1831         server.sofd = anetUnixServer(server.neterr,server.unixsocket,
1832             server.unixsocketperm, server.tcp_backlog);
1833         if (server.sofd == ANET_ERR) {
1834             serverLog(LL_WARNING, "Opening Unix socket: %s", server.neterr);
1835             exit(1);
1836         }
1837         anetNonBlock(NULL,server.sofd);

```

图 1.20 初始化服务器

处理“客户端连接”事件的回调函数也是在 `initServer` 方法中设置的，如图 1.21 所示。

```

/* Create an event handler for accepting new connections in TCP and Unix
 * domain sockets. */
for (j = 0; j < server.ipfd_count; j++) {
    if (aeCreateFileEvent(server.el,server.ipfd[j], AE_READABLE,
        acceptTcpHandler,NULL) == AE_ERR)
    {
        serverPanic(
            "Unrecoverable error creating server.ipfd file event.");
    }
}
if (server.sofd > 0 && aeCreateFileEvent(server.el,server.sofd,AE_READABLE,
    acceptUnixHandler,NULL) == AE_ERR) serverPanic("Unrecoverable error creating server.sofd file event.");

```

图 1.21 设置回调函数

当服务器初始化后，`aeMain` 方法会被调用，如图 1.22 所示。

```

3842     aeSetBeforeSleepProc(server.el,beforeSleep);
3843     aeSetAfterSleepProc(server.el,afterSleep);
3844     aeMain(server.el);
3845     aeDeleteEventLoop(server.el);
3846     return 0;

```

图 1.22 调用 `aeMain` 方法

在 `aeMain` 方法中，`sProcessEvents` 方法被连续地调用来处理各种事件，如图 1.23 所示：

```

459 void aeMain(aeEventLoop *eventLoop) {
460     eventLoop->stop = 0;
461     while (!eventLoop->stop) {
462         if (eventLoop->beforeSleep != NULL)
463             eventLoop->beforeSleep(eventLoop);
464         aeProcessEvents(eventLoop, AE_ALL_EVENTS|AE_CALL_AFTER_SLEEP);
465     }
466 }

```

图 1.23 调用 `sProcessEvents` 方法

1.6.4 更多细节

显而易见，在 `polling` 的过程中没有线程或子进程的创建和交互。因此，该模型的关键优点就在于它是一个轻量级上下文切换的 I/O 模型，在上下文切换上花费的代价不大。当然，这个处理模型也有一些限制需要考虑。我们可能遇到的最常见的问题就是延迟问题。在 `polling` 模型中，在一条命令被处理完成前，Redis 不能处理其他的命令。因此，请从现在开始牢记，在使用 Redis 时，意外的延迟问题将是最让人头痛的。

由于篇幅有限，本书并未讨论诸如 `poll`、`select` 等其他的轮询方法。如果读者是在非 Linux 平台上运行此案例，那么可以通过调试源码来了解有关具体平台上轮询机制的更多细节。

1.6.5 相关内容

- 请参阅使用 `SLOWLOG` 识别慢查询和第 10 章 Redis 的故障诊断中延迟问题的故障诊断一节来学习有关 Redis 延迟故障诊断的更多细节。
- 如果读者想在 Windows 中构建和调试源代码，请参阅附录 A Windows 环境搭建。
- 读者还可以进一步参考以下链接：
 - <https://redis.io/topics/internals-eventlib> 详细地讲解了事件库。
 - <https://redis.io/topics/internals-rediseventlib> 描述了 Redis 事件库的细节。
- 如果读者使用的不是 Ubuntu 操作系统，可以参考 <https://cmake.org/install/> 来安装 CMake。

1.7 理解 Redis 通信协议

正如我们在此前的案例中所讲解的那样，Redis 基本上就是一个接受并处理来自客户端请求的非阻塞、I/O 复用的 TCP 服务器。换句话说，虽然 Redis 服务器很复杂，但我们可以使用各种编程语言通过 TCP 协议与 Redis 进行通信。术语协议（*protocol*）代表网络通信中服务器和客户端之间使用的语言。对于 Redis 来说，这种通信协议叫做 **REdis Serialization Protocol (RESP, Redis 序列化协议)**。在本案例中，我们将学习 RESP 的工作原理。

我们应该首先注意一件事: 虽然乍看上去初学者学习本案例似乎有点太早了; 但是, 我们确实认为将 RESP 作为基础知识来学习并不困难, 而且对读者是有好处的。这是因为在了解了 RESP 后, 去理解那些使用不同编程语言实现的 Redis 客户端和代理对读者来说将不再神秘。

当然, 读者也可以按需跳过本案例, 这也不会影响对后续章节的学习。

1.7.1 准备工作

我们需要按照本章启动和停止 *Redis* 一节中所描述的步骤配置并启动一个 Redis 服务器。

此外, 我们还需要安装 *netcat* (*nc*) 工具。在本案例中, 我们使用了 Ubuntu 16.04.3 LTS 中 *netcat* 提供的 *nc* 命令。如果读者使用的是 Windows 操作系统, 那么可以使用 *Cygwin* 来安装 *netcat*。

1.7.2 操作步骤

接下来, 让我们按照以下的步骤来学习 Redis 通信协议:

1. 使用 *netcat* 向 Redis 服务器发送 *PING* 命令:

请注意, 我们没有通过 *redis-cli* 发送 *PING* 命令, 而是使用 RESP 来构造命令:

```
$ echo -e "*1\r\n$4\r\nPING\r\n" | nc 127.0.0.1 6379
+PONG
```

2. 使用 *SET* 和 *INCR* 命令放置一个整型并将其加 1:

```
$ echo -e "*3\r\n$3\r\nset\r\n$5\r\nmykey\r\n$1\r\n1\r\n" | nc 127.0.0.1
6379
+OK
$ echo -e "*2\r\n$4\r\nINCR\r\n$5\r\nmykey\r\n" | nc 127.0.0.1 6379
:2
```

当发送了不存在的命令时, 我们可能会遇到以下的错误:

```
$ echo -e "*2\r\n$3\r\nget\r\n$3\r\nfoo\r\n" | nc 127.0.0.1 6379
-ERR unknown command 'got'
```

多个命令可以组合在一起并在一次网络传输中发送给 Redis 服务器:

```
$ echo -e "*3\r\n$3\r\nset\r\n$3\r\nfoo\r\n$3\r\nbar\r\n*2\r\n$3\r\nget\r\n$3\r\nfoo\r\n" | nc 127.0.0.1 6379
+OK
$3
```

```
bar
```

1.7.3 工作原理

在开始学习这些命令时，读者很可能会感到迷茫。因为正如我们在本节准备工作中所提到的，这些命令是 Redis 服务器和客户端通信时使用的语言；但是，由于 RESP 只包括五种类型，所以实际学起来并不难。

接下来，让我们来学习每个命令。

首先，我们向 Redis 服务器发送了 `*1\r\n$4\r\nPING\r\n`。该命令开头的星号表示这是一个数组类型。

具体地说：

- 1 代表这个数组的大小。
- `\r\n`（CRLF）是 RESP 中每个部分的终结符。
- `$4` 之前的反斜杠是 `$` 符号的转义字符。
- `$4` 表示接下来是四个字符组成的字符串。
- `PING` 为字符串本身。
- `+PONG` 是 `PING` 命令的响应字符串。加号表示响应是一个简单字符串类型。

接下来，让我们学习整型。`INCR` 命令返回的结果是 `:2`，其中数字前的冒号就代表结果是一个整型。

有时，如果服务器收到了不存在的命令，比如上面演示的 `got` 命令，服务器可能会返回一个以减号开头的错误类型的消息。

另外，出于对性能的考虑，我们可以使用 RESP 在一次对 Redis 服务器的调用中发送多个命令。

总之，客户端发送给 Redis 服务器的命令实际上就是一组字符串组成的 RESP 数组。之后，服务器按照不同的命令、使用上述五种 RESP 类型之一对命令进行响应。

1.7.4 相关内容

- 有关 Redis 协议更详细介绍，请参阅：<https://redis.io/topics/protocol>。
- 在 GitHub 上可以找到很多 Redis 客户端和代理。例如，我们可以在 <https://github.com/tonivade/resp-server> 上找到一个基于 Java Netty 实现的 RESP 协议。
- 另一个例子是 <https://github.com/tidwall/resp>，其中包括了使用 Go 语言编写的 RESP 协议的构造器、解析器和服务端。

第 2 章

数据类型

在本章中，我们将学习下列案例：

- 使用字符串（string）数据类型。
- 使用列表（list）数据类型。
- 使用哈希（hash）数据类型。
- 使用集合（set）数据类型。
- 使用有序集合（sorted set）数据类型。
- 使用 HyperLogLog 数据类型。
- 使用 Geo 数据类型。
- 键管理。

2.1 本章概要

使用 Redis 进行应用设计和开发的一个核心概念是数据类型。与关系数据库不同，在 Redis 中不存在需要我们担心的表或模式。在使用 Redis 进行应用设计和开发时，我们首先应该考虑的是，Redis 原生支持的哪种数据类型最适合我们的场景。此外，我们无法像在关系数据库中那样，使用 SQL 来操作 Redis 中的数据。相反，我们需要直接使用 API 发送数据所对应的命令，来操作想要操作的目标数据。

在本章中，我们将研究与 Redis 相关的所有数据类型及其重要操作。为了更好地说明这些数据类型及其操作，我们将展示一个类似于 Yelp 的示例程序（本书中将其称为 *Relp*）。Relp 是一个供用户评论和推荐优秀餐厅、购物中心或其他服务的应用。在 Relp 中，我们可以浏览一个城市中不同的

餐厅，找到在一定距离范围内排名前十的健身房，给本地服务打分和发表评论意见，等等。我们会把 Relp 所涉及的数据全部存储到 Redis 中。

2.2 使用字符串（string）类型

字符串类型是编程语言和应用程序中最常见和最有用的数据类型，也是 Redis 的基本数据类型之一。事实上，Redis 中所有的键都必须是字符串。本案例将演示在 Redis 中操作字符串的基本命令。

2.2.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

2.2.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用字符串类型。

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。

2. 使用 `SET` 命令将一个字符串值关联到一个键。在 Relp 中，我们可以将餐厅名称和地址分别用作键和值；例如，假设我们想设置“Extreme Pizza”餐厅的地址：

```
127.0.0.1:6379> SET "Extreme Pizza" "300 Broadway, New York, NY"
OK
```

3. 使用 `GET` 命令可以轻松地取回字符串的值：

```
127.0.0.1:6379> GET "Extreme Pizza"
"300 Broadway, New York, NY"
```

4. 当 `GET` 一个不存在的键时会返回 `(nil)`：

```
127.0.0.1:6379> GET "Yummy Pizza"
(nil)
```

5. `STRLEN` 命令返回字符串的长度；例如，如果我们想获取“Extreme Pizza”地址的长度，可以使用：

```
127.0.0.1:6379> STRLEN "Extreme Pizza"
(integer) 26
```

6. 对不存在的键执行 `STRLEN` 命令会返回 0。

Redis 还提供了一些命令来直接操作字符串。使用这些命令的好处是，不需要通过 `GET` 命令来读取一个字符串的值，再用 `SET` 命令将（处理后的）字符串写回去。

- 使用 `APPEND` 命令可以向一个键的字符串值末尾追加字符串：

```
127.0.0.1:6379> APPEND "Extreme Pizza" " 10011"
```

```
(integer) 32
```

```
127.0.0.1:6379> GET "Extreme Pizza"
```

```
"300 Broadway, New York, NY 10011"
```

- 使用 `SETRANGE` 命令可以覆盖字符串值的一部分；例如，如果我们想更新“Extreme Pizza”的地址，可以使用：

```
127.0.0.1:6379> SETRANGE "Extreme Pizza" 14 "Washington, DC 20009"
```

```
(integer) 34
```

```
127.0.0.1:6379> GET "Extreme Pizza"
```

```
"300 Broadway, Washington, DC 20009"
```

2.2.3 工作原理

`SET` 和 `GET` 可能是 Redis 中最常用的命令了。`SET` 命令的用法非常简单：

```
SET <key> <value>
```

如果 `SET` 命令执行成功，Redis 会返回 `OK`。`APPEND` 命令会将字符串追加到现有字符串的末尾，并返回新字符串的长度。如果键不存在，那么 Redis 将首先创建一个空字符串并与键相关联，然后再执行 `APPEND` 命令。`SETRANGE` 命令会覆盖字符串的一部分（从指定的偏移开始，直到整个字符串的末尾）。在 Redis 中，字符串的偏移是从 0 开始的。`SETRANGE` 命令会在覆盖完成后返回新字符串的长度。

2.2.4 更多细节

如果某个键已经存在，那么 `SET` 命令会覆盖该键此前对应的值。有时，我们不希望在键存在的时候盲目地覆盖键；这时，我们可以使用 `EXIST` 命令来测试键的存在性。事实上，Redis 提供了 `SETNX` 命令（简称为不存在时 `SET`），用于原子性地、仅在键不存在时设置键的值。如果键的值设置成功，则 `SETNX` 返回 1；如果键已经存在，则返回 0 且不覆盖原来的值：

```
127.0.0.1:6379> SETNX "Lobster Palace" "437 Main St, Chicago, IL"
```

```
(integer) 1
127.0.0.1:6379> SETNX "Extreme Pizza" "100 Broadway, New York, NY"
(integer) 0
```

SET 命令中的 NX 选项与 SETNX 一样。相反地，SET 命令的 XX 选项表示仅在键已经存在时才设置值。

我们可以通过使用 MSET 和 MGET 命令来一次性地设置和获取多个键的值。使用 MSET 的优点在于整个操作是原子性的，意味着所有的键都是在客户机和服务器之间的一次通信中设置的。因此，我们可以通过使用 MSET 命令而不是发出多个 SET 命令来节省网络开销。MSET 和 MGET 命令的用法为：

```
MSET key value [key value...]
MGET key value [key value...]

127.0.0.1:6379> MSET "Sakura Sushi" "123 Ellis St, Chicago, IL" "Green Curry
    Thai" "456 American Way, Seattle, WA"
OK

127.0.0.1:6379> MGET "Sakura Sushi" "Green Curry Thai" "nonexistent"
1) "123 Ellis St, Chicago, IL"
2) "456 American Way, Seattle, WA"
3) (nil)
```

在这里有必要提一下字符串在 Redis 内部是如何进行编码的。Redis 使用了三种不同的编码方式来存储字符串对象，并会根据每个字符串值自动决定所使用的编码方式：

- int：用于能够使用 64 位有符号整数表示的字符串。
- embstr：用于长度小于或等于 44 字节（在 Redis 3.x 中曾经是 39 字节）的字符串；这种类型的编码在内存使用和性能方面更有效率。
- raw：用于长度大于 44 字节的字符串。

我们可以使用 OBJECT 命令来查看与键相关联的 Redis 值对象的内部编码方式：

```
127.0.0.1:6379> SET myKey 12345
OK
127.0.0.1:6379> OBJECT ENCODING myKey
"int"
127.0.0.1:6379> SET myKey "a string"
OK
```



```
127.0.0.1:6379> OBJECT ENCODING myKey
"embstr"
127.0.0.1:6379> SET myKey "a long string whose length is more than 44 bytes"
OK
127.0.0.1:6379> OBJECT ENCODING myKey
"raw"
```

2.2.5 相关内容

- OBJECT 命令除了用于查看编码外还有更多的功能，该命令还允许我们查看 Redis 对象的 refcount 和 idletime（参见<https://redis.io/commands/object>）。
- 受篇幅所限，在本案例中我们无法演示字符串类型相关的所有 Redis 命令。请读者参阅<https://redis.io/commands#string>学习字符串相关的所有命令。
- 请参阅本章键管理一节学习键的管理，包括获取键的列表、重命名键和删除键。

2.3 使用列表（list）类型

列表是应用程序开发中非常有用的数据类型之一。列表能够存储一组对象，因此它也可以被用作栈或者队列。在 Redis 中，与键相关联的值可以是字符串组成的列表。Redis 中的列表更像是数据结构世界中的双向链表。本案例将演示 Redis 中操作列表的基本命令。

2.3.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 redis-cli 连接到这个 Redis 服务器。

2.3.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用列表类型：

1. 打开一个终端，并使用 redis-cli 连接到 Redis。
2. 在这里，我们将使用一个列表来存储用户最喜欢的餐厅。使用 LPUSH 在列表的左端插入两个餐厅的名称：

```
127.0.0.1:6379> LPUSH favorite_restaurants "PF Chang's" "Olive Garden"
(integer) 2
```

3. 如果要获取列表中所有餐厅的名称，可以使用 `LRANGE`：

```
127.0.0.1:6379> LRANGE favorite_restaurants 0 -1
1) "Olive Garden"
2) "PF Chang's"
```

4. 如果要在列表的右端插入餐厅的名称，可以使用 `RPUSH`：

```
127.0.0.1:6379> RPUSH favorite_restaurants "Outback Steakhouse" "Red Lobster"
(integer) 4
127.0.0.1:6379> LRANGE favorite_restaurants 0 -1
1) "Olive Garden"
2) "PF Chang's"
3) "Outback Steakhouse"
4) "Red Lobster"
```

5. 如果要在“PF Chang's”之后插入一个新餐厅的名称，可以使用 `LINSERT`：

```
127.0.0.1:6379> LINSERT favorite_restaurants AFTER "PF Chang's" "Indian
Tandoor"
(integer) 5
127.0.0.1:6379> LRANGE favorite_restaurants 0 -1
1) "Olive Garden"
2) "PF Chang's"
3) "Indian Tandoor"
4) "Outback Steakhouse"
5) "Red Lobster"
```

6. 如果要获取列表中位于索引位置 3 处餐厅的名称，可以使用 `LINDEX`：

```
127.0.0.1:6379> LINDEX favorite_restaurants 3
"Outback Steakhouse"
```

2.3.3 工作原理

正如我们此前所提到的，Redis 中的列表与双向链表类似，因此可以使用以下的三个命令将新元素添加到列表中：

- `LPUSH`：将元素添加到列表的左端。
- `RPUSH`：将元素添加到列表的右端。

- LINSERT: 将元素插入到列表的支点/枢轴元素 (pivot element) 之前或之后。

LPUSH、RPUSH 和 LINSERT 会返回插入后列表的长度。在向列表中插入元素前，无需为一个键初始化一个空列表。如果我们向一个不存在的键中插入元素，Redis 将首先创建一个空列表并将其与键关联。同样，我们也不需要删除值为空列表的键，因为 Redis 会为我们回收这种键。

提示

如果我们仅想在列表存在时才将元素插入到列表中，那么可以使用 LPUSHX 和 RPUSHX 命令。

我们可以使用 LPOP 或 RPOP 从列表中删除一个元素。这两个命令会从列表的左端或右端移除第一个元素，并返回其值。当对不存在的键执行 LPOP 或 RPOP 时，将返回 (nil):

```
127.0.0.1:6379> LPOP favorite_restaurants
"Olive Garden"
127.0.0.1:6379> RPOP favorite_restaurants
"Red Lobster"
127.0.0.1:6379> LPOP non_existent
(nil)
```

我们可以使用 LINDEX 命令从列表中获取位于指定索引处的元素，也可以使用 LRANGE 命令获取一个范围内的元素。

提示

在 Redis 中，列表索引是怎么定义的呢？假设列表中有 N 个元素；列表的索引可以按照从左到右的方式指定为 $0 \sim N-1$ ，也可以按照从右到左的方式指定为 $-1 \sim -N$ 。因此， $0 \sim -1$ 就表示整个列表。Redis 列表索引的使用方法与 Python 列表索引的使用方法非常类似，而 Python 列表与其他编程语言中的数组更像。

LTRIM 命令可用于在删除列表中的多个元素的同时，只保留由 *start* 和 *end* 索引所指定范围内的元素：

```
127.0.0.1:6379> LRANGE favorite_restaurants 0 -1
1) "PF Chang's"
2) "Indian Tandoor"
3) "Outback Steakhouse"
127.0.0.1:6379> LTRIM favorite_restaurants 1 -1
OK
```



```
127.0.0.1:6379> LRange favorite_restaurants 0 -1
```

```
1) "Indian Tandoor"
```

```
2) "Outback Steakhouse"
```

我们可以使用 `LSET` 命令设置列表中指定索引位置处元素的值：

```
127.0.0.1:6379> LSET favorite_restaurants 1 "Longhorn Steakhouse"
```

```
OK
```

```
127.0.0.1:6379> LRange favorite_restaurants 0 -1
```

```
1) "Indian Tandoor"
```

```
2) "Longhorn Steakhouse"
```

2.3.4 更多细节

`LPOP` 和 `RPOP` 命令有对应的阻塞版本：`BLPOP` 和 `BRPOP`。与非阻塞版本类似，阻塞版本的命令也从列表的左端或右端弹出元素；但是，当列表为空时，阻塞版本会将客户端阻塞。我们必须在这些阻塞版本的命令中指定一个以秒为单位的超时时间，表示最长等待几秒。当超时时间为零时，表示永久等待。这个特性在任务调度场景中非常有用；在这种场景下，多个任务执行程序（Redis 客户端）会等待任务调度程序分配新的任务。任务执行程序只需要对 Redis 中的列表使用 `BLPOP` 或 `BRPOP`，之后每当有新任务时，调度程序把任务插入到列表中，而任务执行程序之一便会获取到该任务。

让我们再打开两个终端，它们代表两个 Redis 客户端，`worker-1` 和 `worker-2`，然后使用 `redis-cl` 连接到同一个 Redis 服务器。假设之前的那个 Redis 客户端是任务调度程序。

我们从两个任务执行器对应的终端上对同一个列表 `job_queue` 执行 `BRPOP` 命令，让这两个任务执行器在同一个队列上等待新的任务：

```
worker-1> BRPOP job_queue 0
```

```
worker-2> BRPOP job_queue 0
```

然后，从任务调度程序对应的终端上把一个新元素插入到列表中：

```
dispatcher> LPUSH job_queue job1
```

```
(integer) 1
```

由于 `worker-1` 是在 `worker-2` 之前执行的 `BRPOP`，所以 `worker-1` 先被解除阻塞状态并得到了 `job1`：

```
worker-1> BRPOP job_queue 0
```

```
1) "job_queue"
```

```
2) "job1"
(170.81s)
```

此时，worker-2 仍然被阻塞。让我们从任务调度程序对应的终端上再向列表中多添加两个元素：

```
dispatcher> LPUSH job_queue job2 job3
```

worker-2 的阻塞状态被解除并得到了 job2，而 job3 则留在了列表中等待被获取：

```
worker-2> BRPOP job_queue 0
1) "job_queue"
2) "job2"
(358.12s)
```

```
dispatcher> LRANGE job_queue 0 -1
1) "job3"
```

Redis 在内部使用 quicklist 存储列表对象。有两个配置选项可以调整列表对象的存储逻辑：

- `list-max-ziplist-size`：一个列表条目中一个内部节点的最大大小（译者注：quicklist 的每个节点都是一个 ziplist）。在大多数情况下，使用默认值即可。
- `list-compress-depth`：列表压缩策略。如果我们会用到 Redis 中列表首尾的元素，那么可以利用这个选项来获得更好的压缩比（译者注：该参数表示 quicklist 两端不被压缩的节点的个数，当列表很长的时候最可能被访问的数据是位于列表两端的数据，因此对这个参数精确地进行调优可以实现在压缩比和其他因素之间的平衡）。

2.3.5 相关内容

- 我们没有覆盖列表类型相关的所有 Redis 命令。请读者参阅<https://redis.io/commands#list>学习列表相关的所有命令。

2.4 使用哈希（hash）类型

哈希表示字段和值之间的映射关系，与某些编程语言中的 map 或字典类型类似。Redis 数据集本身就可以被看作一个哈希，其中字符串类型的键关联到诸如字符串和列表之类的数据对象（就像我们在前两个案例中所看到的那样）。而 Redis 的数据对象也可以再次使用哈希，其字段和值必须是字符串类型。为了与 Redis 的键进行区分，我们使用“字段（field）”来表示哈希中值对象所关联的

键。哈希对于存储对象属性而言是一种完美的数据类型。在本案例中，我们将使用哈希来存储餐厅的信息，例如地址、电话号码和评分等。

2.4.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

2.4.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用哈希类型。

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 接下来，让我们使用 `HMSET` 命令来设置“Kyoto Ramen”餐厅的属性信息：

```
127.0.0.1:6379> HMSET "Kyoto Ramen" "address" "801 Mission St, San Jose, CA" "
phone" "555-123-6543" "rating" "5.0"
OK
```

3. 使用 `HMGET` 命令从一个哈希中获取多个字段对应的值：

```
127.0.0.1:6379> HMGET "Kyoto Ramen" "address" "phone" "rating"
1) "801 Mission St, San Jose, CA"
2) "555-123-6543"
3) "5.0"
```

4. 使用 `HGET` 命令从一个哈希中获取某个字段对应的值：

```
127.0.0.1:6379> HGET "Kyoto Ramen" "rating"
"5.0"
```

5. 使用 `HEXISTS` 命令测试一个哈希中是否存在某个字段：

```
127.0.0.1:6379> HEXISTS "Kyoto Ramen" "phone"
(integer) 1
127.0.0.1:6379> HEXISTS "Kyoto Ramen" "hours"
(integer) 0
```

6. 使用 `HGETALL` 命令获取一个哈希中的所有字段和值：


```
127.0.0.1:6379> HGETALL "Kyoto Ramen"
```

```
1) "address"
```

```
2) "801 Mission St, San Jose, CA"
```

```
3) "phone"
```

```
4) "555-123-6543"
```

```
5) "rating"
```

```
6) "5.0"
```

提示

不建议对数量巨大的哈希使用 `HGETALL`；具体原因将在稍后进行解释。

7. 类似地，我们可以使用 `HSET` 命令设置单个字段的值。此命令可用于修改现有字段的值或添加新的字段：

```
127.0.0.1:6379> HSET "Kyoto Ramen" "rating" "4.9"
```

```
(integer) 0
```

```
127.0.0.1:6379> HSET "Kyoto Ramen" "status" "open"
```

```
(integer) 1
```

```
127.0.0.1:6379> HMGET "Kyoto Ramen" "rating" "status"
```

```
1) "4.9"
```

```
2) "open"
```

8. 使用 `HDEL` 命令从哈希中删除字段：

```
127.0.0.1:6379> HDEL "Kyoto Ramen" "address" "phone"
```

```
(integer) 2
```

```
127.0.0.1:6379> HGETALL "Kyoto Ramen"
```

```
1) "rating"
```

```
2) "4.9"
```

2.4.3 工作原理

与我们在列表（*list*）类型一节中所提到的类似，我们不需要在添加字段前先初始化一个空的哈希。Redis 会自动使用 `HSET` 和 `HMSET` 来实现这一点。类似地，当哈希变成空的时，Redis 会负责将其删除。

默认情况下，HSET 和 HMSET 会覆盖现有的字段。HSETNX 命令则仅在字段不存在的情况下才设置字段的值，可用于防止 HSET 的默认覆盖行为：

```
127.0.0.1:6379> HSETNX "Kyoto Ramen" "phone" "555-555-0001"
(integer) 0
127.0.0.1:6379> HGET "Kyoto Ramen" "phone"
"555-123-6543"
```

对于不存在的键或字段，HMGET 和 HGET 将返回 (nil)：

```
127.0.0.1:6379> HMGET "Kyoto Ramen" "rating" "hours"
1) "4.9"
2) (nil)

127.0.0.1:6379> HGET "Little Sheep Mongolian" "address"
(nil)
```

2.4.4 更多细节

一个哈希最多能够容纳 2^{32} -1 个字段。如果一个哈希的字段非常多，那么执行 HGETALL 命令时可能会阻塞 Redis 服务器。在这种情况下，我们可以使用 HSCAN 命令来增量地获取所有字段和值。

HSCAN 是 Redis 中 SCAN 命令的一种（SCAN、HSCAN、SSCAN、ZSCAN），该命令会增量地迭代遍历元素，从而不会造成服务器阻塞。HSCAN 命令是一种基于指针的迭代器，因此我们需要在每次调用命令时指定一个游标（从 0 开始）。当一次 HSCAN 运行结束后，Redis 将返回一个元素列表以及一个新的游标，这个游标可以用于下一次迭代。

HSCAN 的使用方法如下：

- HSCAN key cursor [MATCH pattern] [COUNT number]。
- 选项 MATCH 可以用来匹配满足指定 Glob 表达式的字段。
- 选项 COUNT 用来说明在每次迭代中应该返回多少个元素。但是，这个选项仅仅是一种参考，Redis 并不保证返回的元素数量就是 COUNT 个。COUNT 的默认值是 10。

假设我们有一个非常大的哈希，其中有数百万个甚至更多的字段。下面，让我们使用 HSCAN 来遍历包含关键字 garden 的字段：

```
127.0.0.1:6379> HSCAN restaurant_ratings 0 MATCH *garden*
1) "309"
2) 1) "panda garden"
```

- 2) "3.9"
- 3) "chang's garden"
- 4) "4.5"
- 5) "rice garden"
- 6) "4.8"

我们可以使用由服务器返回的新游标 309 来进行一次新的迭代：

```
127.0.0.1:6379> HSCAN restaurant_ratings 309 MATCH *garden*
1) "0"
2) 1) "szechuwan garden"
    2) "4.9"
    3) "garden wok restaurant"
    4) "4.7"
    5) "win garden"
    6) "4.0"
    7) "east garden restaurant"
    8) "4.6"
```

请注意，当服务器返回的新游标为 0 时，意味着整个遍历完成。

Redis 在内部使用两种编码来存储哈希对象：

- **ziplist**：对于那些长度小于配置中 `hash-max-ziplist-entries` 选项配置的值（默认为 512），且所有元素的大小都小于配置中 `hash-max-ziplist-value` 选项配置的值（默认为 64 字节）的哈希，采用此编码。ziplist 编码对于较小的哈希而言可以节省占用空间。
- **hashtable**：当 ziplist 不适用时使用的默认编码。

2.4.5 相关内容

- 有关 SCAN 命令的更多细节，请参阅：<https://redis.io/commands/scan>。

2.5 使用集合（set）类型

集合类型是由唯一、无序对象组成的集合（collection）。它经常用于测试某个成员是否在集合中、重复项删除和集合运算（求并、交、差集）。Redis 的值对象可以是字符串集合。在本案例中，我们会将餐厅的标签保存在 Redis 中，并演示 Redis 用于集合操作的基本命令。

2.5.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

2.5.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用集合类型。

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 使用 `SADD` 命令给 “Original Buffalo Wings” 餐厅添加标签：

```
127.0.0.1:6379> SADD "Original Buffalo Wings" "affordable" "spicy" "busy" "
    great taste"
(integer) 4
```

3. 使用 `SISMEMBER` 测试一个元素是否位于集合中：

```
127.0.0.1:6379> SISMEMBER "Original Buffalo Wings" "busy"
(integer) 1
127.0.0.1:6379> SISMEMBER "Original Buffalo Wings" "costly"
(integer) 0
```

4. 使用 `SREM` 命令从集合中删除元素，例如，让我们从餐厅的标签中删除 “busy” 和 “spicy”：

```
127.0.0.1:6379> SREM "Original Buffalo Wings" "busy" "spicy"
(integer) 2
127.0.0.1:6379> SISMEMBER "Original Buffalo Wings" "busy"
(integer) 0
127.0.0.1:6379> SISMEMBER "Original Buffalo Wings" "spicy"
(integer) 0
```

5. 使用 `SCARD` 命令获取集合中成员的数量：

```
127.0.0.1:6379> SCARD "Original Buffalo Wings"
(integer) 2
```

2.5.3 工作原理

与列表和哈希类似，Redis 在执行 `SADD` 命令时，如果键不存在就会为我们创建一个空集合；同样，Redis 也会自动地删除空集合所对应的键。

2.5.4 更多细节

在 Redis 中，一个集合最多可以容纳 $2^{23} - 1$ 个成员。

我们可以使用 `SMEMBERS` 命令列出集合中的所有元素。但是，与我们在使用哈希（*hash*）类型中所提到的 `HGETALL` 类似，在一个大的集合中使用 `SMEMBERS` 命令可能会阻塞服务器。因此，我们并不推荐使用 `SMEMBERS` 命令，而是应该使用 `SSCAN` 命令。`SSCAN` 与我们在使用哈希（*hash*）类型中介绍的 `HSCAN` 命令的用法非常类似。

Redis 提供了一组集合运算相关的命令，`SUNION` 和 `SUNIONSTORE` 用于计算并集，`SINTER` 和 `SINTERSTORE` 用于计算交集，`SDIFF` 和 `SDIFFSTORE` 用于计算差集。不带 `STORE` 后缀的命令只返回相应操作的结果集合，而带 `STORE` 后缀的命令则会将结果存储到一个指定的键中。

再举一个使用 `SINTER` 和 `SINTERSTORE` 命令的例子。让我们给另一家餐厅 "Big Bear Wings" 添加标签，然后获取 "Original Buffalo Wings" 和 "Big Bear Wings" 餐厅共有的标签：

```
127.0.0.1:6379> SMEMBERS "Original Buffalo Wings"
1) "affordable"
2) "great taste"
127.0.0.1:6379> SADD "Big Bear Wings" "affordable" "spacious" "great music"
(integer) 3
127.0.0.1:6379> SINTER "Original Buffalo Wings" "Big Bear Wings"
1) "affordable"
127.0.0.1:6379> SINTERSTORE "common_tags" "Original Buffalo Wings" "Big Bear Wings"
(integer) 1
127.0.0.1:6379> SMEMBERS "common_tags"
1) "affordable"
```

Redis 在内部使用两种编码方式来存储集合对象：

- `intset`：对于那些元素都是整数，且元素个数小于配置中 `set-max-intset-entries` 选项设置的值（默认 512）的集合，采用此编码。`intset` 编码对于较小的集合而言可以节省占用空间。
- `hashtable`：`intset` 不适用时的默认编码。

2.6 使用有序集合（sorted set）类型

与此前案例中所介绍的集合相比，有序集合是一个类似但更复杂的数据类型。单词“Sorted”意味着这种集合中的每个元素都拥有一个可用于排序的权重，并且我们可以按顺序从集合中获取元素。在某些需要一直保持数据有序的场景中，使用这种原生的有序特性是很方便的。在本案例中，我们将学习如何操作有序集合。

2.6.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 *Redis* 服务器，并使用 `redis-cli` 连接到这个 *Redis* 服务器。

2.6.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用有序集合类型。

1. 打开一个终端，并使用 `redis-cli` 连接到 *Redis*。

2. 按以下步骤对 *Relp* 中的当地餐厅进行排序：

(1) 使用 `ZADD` 命令将点评数和每个餐厅的名字放入一个有序集合中：

```
127.0.0.1:6379> ZADD ranking:restaurants 100 "Olive Garden" 23 "PF Chang's" 34
                    "Outback Steakhouse" 45 "Red Lobster" 88 "Longhorn Steakhouse"
(integer) 5
```

(2) 然后使用 `ZREVRANGE` 命令来获取这个排名：

```
127.0.0.1:6379> ZREVRANGE ranking:restaurants 0 -1 WITHSCORES
1) "Olive Garden"
2) "100"
3) "Longhorn Steakhouse"
4) "88"
5) "Red Lobster"
6) "45"
7) "Outback Steakhouse"
8) "34"
9) "PF Chang's"
10) "23"
```

(3) 如果 *Relp* 中的某个用户点了赞，那么我们可以使用命令 `ZINCRBY` 对餐厅的投票数加一：


```
127.0.0.1:6379> ZINCRBY ranking:restaurants 1 "Red Lobster"  
"46"
```

(4) 如果某个用户想要浏览某个特定餐厅的排名和投票数,那么可以使用 ZREVRANK 和 ZSCORE 命令:

```
127.0.0.1:6379> ZREVRANK ranking:restaurants "Olive Garden"  
(integer) 0
```

```
127.0.0.1:6379> ZSCORE ranking:restaurants "Olive Garden"  
"100"
```

(5) 当出现了另一个来自不同数据源的、更靠谱的餐厅排名时,如果我们需要合并这两个排名的话,那么可以使用 ZUNIONSTORE 命令:

```
127.0.0.1:6379> ZADD ranking2:restaurants 50 "Olive Garden" 33 "PF Chang's" 55  
"Outback Steakhouse" 190 "Kung Pao House"  
(integer) 4
```

```
127.0.0.1:6379> ZUNIONSTORE totalranking 2 ranking:restaurants ranking2:  
restaurants WEIGHTS 1 2  
(integer) 6
```

```
127.0.0.1:6379> ZREVRANGE totalranking 0 -1 WITHSCORES  
1) "Kung Pao House"  
2) "380"  
3) "Olive Garden"  
4) "200"  
5) "Outback Steakhouse"  
6) "144"  
7) "PF Chang's"  
8) "89"  
9) "Longhorn Steakhouse"  
10) "88"  
11) "Red Lobster"  
12) "46"  
127.0.0.1:6379>
```

2.6.3 工作原理

在 ZADD 命令中使用 NX 选项，能够实现在不更新已存在成员的情况下只添加新的成员：

```
127.0.0.1:6379> ZADD ranking:restaurants NX 50 "Olive Garden"
```

```
(integer) 0
```

```
127.0.0.1:6379> ZREVRANGE ranking:restaurants 0 -1 WITHSCORES
```

```
1) "Kung Pao House"
```

```
2) "213"
```

```
3) "Olive Garden"
```

```
4) "100"
```

```
5) "Longhorn Steakhouse"
```

```
6) "88"
```

```
7) "Red Lobster"
```

```
8) "46"
```

```
9) "Outback Steakhouse"
```

```
10) "34"
```

```
11) "PF Chang's"
```

```
12) "23"
```

类似地，选项 xx 允许我们在不向集合中增加新元素的情况下更新集合（译者注：即只更新存在的成员而不添加新成员）。请注意，这些选项只适用于 Redis 3.0.2 或更高版本。

另外，有关 ZADD 还应该注意的是，多个不同的成员可能具有相同的权重。在这种情况下，Redis 将按照字典顺序进行排序。

ZUNIONSTORE 命令用于将两个有序集合的并集保存到指定的键中，且可以指定各个有序集合的不同权重。ZUNIONSTORE 命令的用法如下：

```
ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [
  AGGREGATE SUM|MIN|MAX]
```

ZREVRANGE 命令的开始和结束索引遵循使用哈希（hash）类型一节中所提到的 Redis 索引使用惯例。因此，0 表示第一个元素，1 表示第二个元素，-1 表示最后一个元素，-2 表示倒数第二个元素，依此类推。因此，我们可以使用以下的命令来获取排名前三的餐厅：

```
127.0.0.1:6379> ZREVRANGE totalranking 0 2 WITHSCORES
```

```
1) "Kung Pao House"
```

```
2) "380"
```

- 3) "Olive Garden"
- 4) "200"
- 5) "Outback Steakhouse"
- 6) "144"

2.6.4 更多细节

当涉及复杂的 Redis API 时，我们应该始终关注命令所需的耗时和复杂性。对有序集合而言，ZINTERSTORE 和 ZUNIONSTORE 命令就是这样的情况。

Redis 在内部使用两种编码方式存储有序集合对象：

- **ziplist**: 对于那些长度小于配置中 `zset-max-ziplist-entries` 选项配置的值（默认为 128），且所有元素的大小都小于配置中 `zset-max-ziplist-value` 选项配置的值（默认为 64 字节）的有序集合，采用此编码。ziplist 用于节省较小集合所占用的空间。
- **skiplist**: 当 ziplist 不适用时使用的默认编码。

2.6.5 相关内容

- 受篇幅所限，在本案例中我们无法演示有序集合类型相关的所有 Redis 命令。请读者参阅 https://redis.io/commands#sorted_set 学习有序集合相关的所有命令。

2.7 使用 HyperLogLog 类型

在日常的各种数据处理场景中，“唯一计数”是一项常见的任务。在 Redis 中，虽然我们可以使用集合来进行唯一计数；但是，当数据量增大到上千万时，就需要考虑内存消耗和性能下降问题了。如果我们不需要获取数据集的内容，而只是想得到不同值的个数，那么就可以使用 **HyperLogLog**（**HLL**）数据类型来优化使用集合类型时存在的内存和性能问题。在本案例中，我们将学习如何在 Redis 中使用 HLL。

2.7.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

2.7.2 操作步骤

接下来，让我们可以按照以下的步骤来学习如何使用 HyperLogLog 类型：

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 若要计算 `Relp` 中到访名为 `Olive Garden` 的餐厅的不同客户数，我们可以通过 `PFADD` 命令把用户 ID 加入到一个 HyperLogLog 中：

```
127.0.0.1:6379> PFADD "Counting:Olive Garden" "0000123"
(integer) 1
```

```
127.0.0.1:6379> PFADD "Counting:Olive Garden" "0023992"
(integer) 1
```

3. 然后，使用 `PFCOUNT` 命令获取该餐厅的不同到访者数量：

```
127.0.0.1:6379> PFCOUNT "Counting:Olive Garden"
(integer) 2
```

再举一个更加复杂的例子，如果想要展示 `Olive Garden` 在一个星期中的独立访客数，并将其作为每周流行度的标志。那么，我们可以每天生成一个 HLL，然后用 `PFMERGE` 命令把这 7 天的数据合并为一个：

```
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170903" "0023992" "0023991"
"0045992"
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170904" "0023992" "0023991"
"0045992"
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170905" "0024492" "0023211"
"0045292"
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170906" "0023999" "0063991"
"0045922"
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170907" "0023292" "0023991"
"0045921"
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170908" "0043282" "0023984"
"0045092"
```

```
(integer) 1
127.0.0.1:6379> PFADD "Counting:Olive Garden:20170909" "0023992" "0023991"
"0045992"
(integer) 1

127.0.0.1:6379> PFMERGE "Counting:Olive Garden:20170903week" "Counting:Olive
Garden:20170903" "Counting:Olive Garden:20170904" "Counting:Olive Garden
:20170905" "Counting:Olive Garden:20170906" "Counting:Olive Garden
:20170907" "Counting:Olive Garden:20170908" "Counting:Olive Garden
:20170909"
OK

127.0.0.1:6379> PFCOUNT "Counting:Olive Garden:20170903week"
(integer) 14
```

2.7.3 工作原理

HLL 类型相关的所有命令都是以 PF 开头的，用来向 HLL 数据结构的发明者 Philippe Flajolet 致敬。我们不会在本书中深入探讨 HLL 算法的细节。Reids 中 HLL 的优势在于能够使用固定数量的内存（每个 HyperLogLog 类型的键只需要占用 12KB 内存，却可以计算最多 2^{64} 个不同元素的基数）和常数时间复杂度（每个键 $O(1)$ ）进行唯一计数。不过，由于 HLL 算法返回的基数（译者注：HLL 实际是 cardinality counting 基数计数的一种）可能不准确（标准差小于 1%），因此在决定是否使用 HLL 时需要进行权衡。

2.7.4 更多细节

HLL 实际上是被当做字符串存储的。因此，作为一个键值对，它可以很容易地被持久化至外部或从外部持久化中恢复。

在 Redis 内部，使用了两种方式来存储 HLL 对象：

- **稀疏（Sparse）**：对于那些长度小于配置中 `hll-sparse-max-bytes` 选项设置的值（默认为 3000）的 HLL 对象，采用此编码。稀疏表示方式的存储效率更高，但可能会消耗更多的 CPU 资源。
- **稠密（Dense）**：当稀疏方式不能适用时的默认编码。

2.7.5 相关内容

- 关于 HLL 算法更详细的说明，请参考如下的论文：

Flajolet P, Fusy É, Gandouet O, et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm[C]//AofA: Analysis of Algorithms. Discrete Mathematics and Theoretical Computer Science, 2007: 137-156.

<http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>

2.8 使用 Geo 类型

随着智能手机的不断普及，基于地理位置的服务变得越来越受欢迎。Redis 从 3.2 版本开始正式引入了 Geo（地理位置）相关的 API，用于支持存储和查询这些地理位置相关场景中的坐标。在本案例中，我们将学习 Redis 中的 Geo 数据类型。

2.8.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

2.8.2 操作步骤

接下来，让我们按照以下的步骤来学习如何使用 Geo 类型。

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 对于示例程序 `Relp`，我们可以使用 `GEOADD` 命令把位于加州的五家餐厅增加到名为 `restaurants:CA` 的 Geo 集合中：

```
127.0.0.1:6379> GEOADD restaurants:CA -121.896321 37.916750 "Olive Garden"
-117.910937 33.804047 "P.F. Chang's" -118.508020 34.453276 "Outback
Steakhouse" -119.152439 34.264558 "Red Lobster" -122.276909 39.458300 "
Longhorn Charcoal Pit"
(integer) 5
```

3. 接下来，让我们使用 `GEOPOS` 命令从 Geo 集合中获取指定成员的坐标：

```
127.0.0.1:6379> GEOPOS restaurants:CA "Red Lobster"
1) 1) "-119.1524389386177063"
2) "34.26455707283378871"
```

4. 假设读者位于在 Mount Diablo State Park，其经/纬度是 -121.923170/37.878506，如果读者想知道距当前位置 5km 以内的餐厅，那么可以使用：


```
127.0.0.1:6379> GEORADIUS restaurants:CA -121.923170 37.878506 5 km
```

```
1) "Olive Garden"
```

5. 有时，我们可能需要比较两家餐厅之间的距离；此时，可以使用 GEODIST 命令：

```
127.0.0.1:6379> GEODIST restaurants:CA "P.F. Chang's" "Outback Steakhouse" km
```

```
"90.7557"
```

6. GEORADIUSBYMEMBER 命令与 GEORADIUS 命令非常相似，都可以用来找出位于指定范围内的成员；但是，GEORADIUSBYMEMBER 命令的中心点是由 Geo 集合中的成员决定的，而不是像 GEORADIUS 那样使用输入的经纬度来决定的。例如，使用 GEORADIUSBYMEMBER 命令，我们可以搜索 Geo 集合中距离 “Outback Steakhouse” 距离小于 100 km 的餐厅，而 “Outback Steakhouse” 本身就是 Geo 集合的成员之一：

```
127.0.0.1:6379> GEORADIUSBYMEMBER restaurants:CA "Outback Steakhouse" 100 km
```

```
1) "Red Lobster"
```

```
2) "Outback Steakhouse"
```

```
3) "P.F. Chang's"
```

2.8.3 工作原理

当通过 GEOADD 设置坐标时，这些坐标会被内部转换成一个 52 位的 GEOHASH。GEOHASH 是一个被广泛接受的地理坐标编码系统（Geo-encoding system）。我们需要注意的是，存储在 Geo 中的坐标和 GEOPOS 命令返回的坐标之间可能存在细微的差别。因此，我们不应该期望这两者完全相同。

在 GEORADIUS 和 GEORADIUSBYMEMBER 命令中，我们可以使用 WITHDIST 选项来得到距离，使用 ASC/DESC 选项来控制返回结果的升序或降序。此外，通过 STORE/STOREDIST 选项还可以将 GEORADIUS 和 GEORADIUSBYMEMBER 返回的结果存储到 Redis 中的另一个 Geo 集合中。

2.8.4 更多细节

Geo 集合实际上被存储为一个有序集合（Redis 中的 zset），因此有序集合支持的所有命令都可以用于 Geo 数据类型。例如，我们可以使用 ZREM 从 Geo 集合中移除成员，也可以使用 ZRANGE 来获取 Geo 集合的所有成员。

GEOHASH 的实现是基于一种 52 位整数的表示（实现了低于 1 米的精度）。当需要一个标准的 GEOHASH 字符串时，我们可以使用 GEOHASH 命令来获取一个长度为 11 的字符串。

在性能问题方面，根据 Redis 的文档，GEORADIUS 的时间复杂度为 $O(N+\log(M))$ ，其中 N 为由中心点和半径所决定的圆形区域的外接矩形中成员的个数。因此，如果我们想要获得优秀的性能，就必须记住在一次查询中将半径参数设置得尽可能的小，以覆盖尽可能少的点。

2.8.5 相关内容

- 请参阅<https://en.wikipedia.org/wiki/Geohash>，学习有关 GEOHASH 使用和设计原理相关的内容。
- 请参阅<https://redis.io/commands#geo>，学习有关 Geo 数据结构的所有命令。

2.9 键管理

在本章中，到目前为止，我们已经学习了 Redis 中的所有数据类型。一般来说，Redis 中的数据都是由键值对组成的。因此，管理键是应用程序开发和 Redis 管理的另一个基本知识。在本案例中，我们将介绍键的管理。

2.9.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

为了能清晰地展示对键进行的操作，我们首先使用 `fake2db` 将一些测试数据导入到 Redis 中，步骤如下：

1. 安装 `fake2db` 和 Python Redis driver:

```
$ sudo pip install redis fake2db
Flush all the data of the Redis server:
$ bin/redis-cli flushall
```

2. 将测试数据导入到 Redis 服务器中:

```
$ fake2db --rows 10000 --db redis
2017-09-17 16:44:39,393 gnuhpc Rows argument : 10000
2017-09-17 16:44:46,808 gnuhpc simple_registration Commits are successful
after write job!
2017-09-17 16:45:10,151 gnuhpc detailed_registration Commits are successful
after write job!
```

```
2017-09-17 16:45:47,224 gnuhpc    companies Commits are successful after write
job!
2017-09-17 16:45:47,919 gnuhpc    user_agent Commits are successful after write
job!
2017-09-17 16:46:15,696 gnuhpc    customer Commits are successful after write
job!
```

2.9.2 操作步骤

接下来，让我们按照以下的步骤来学习如何进行键的管理：

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 我们可以通过以下的操作来获取 Redis 中键的个数：

```
127.0.0.1:6379> DBSIZE
(integer) 50000
```

3. 如果我们想获取 Redis 服务器中所有的键，可以使用两种 API。其一是 `KEYS`：

```
127.0.0.1:6379> KEYS *
1) "detailed_registration:8001"
2) "company:3859"
3) "user_agent:4820"
4) "detailed_registration:9330"
...
50000) "company:2947"
(9.30s)
```

4. 另一个是 `SCAN`：

```
127.0.0.1:6379> scan 0
1) "20480"
2) 1) "detailed_registration:8001"
    2) "company:3859"
    3) "company:3141"
    4) "detailed_registration:9657"
    5) "user_agent:2325"
    6) "company:1545"
```



```
7) "company:2521"
8) "detailed_registration:1253"
9) "user_agent:1499"
10) "user_agent:3827"
127.0.0.1:6379> scan 20480
1) "26624"
2) 1) "detailed_registration:5263"
   2) "user_agent:2605"
   3) "detailed_registration:1316"
   4) "user_agent:1683"
   5) "customer:894"
   6) "simple_registration:6411"
   7) "company:3638"
   8) "detailed_registration:1665"
   9) "customer:9344"
  10) "company:7028"
```

....

5. 在某些情况下，我们可能需要删除 Redis 中的键，这可以使用 DEL 命令或 UNLINK 命令（译者注：UNLINK 在 Redis 4.0 以上版本引入，主要用于执行大 KEY 的异步删除，后详）：

```
127.0.0.1:6379> DEL "detailed_registration:1665" "simple_registration:6411" "
user_agent:1683"
(integer) 3
127.0.0.1:6379> UNLINK "company:1664"
(integer) 1
```

6. 要判断一个键是否存在，可以使用 EXISTS 命令：

```
127.0.0.1:6379> EXISTS "simple_registration:7681"
(integer) 1
127.0.0.1:6379> EXISTS "simple_registration:99999"
(integer) 0
```

7. 我们可以使用 TYPE 命令获取键的数据类型：

```
127.0.0.1:6379> TYPE "company:3859"
hash
```

8. 我们可以使用 RENAME 命令来重命名一个键：

```
127.0.0.1:6379> RENAME "customer:6591" "customer:6591:renamed"
```

```
OK
```

2.9.3 工作原理

Redis 中的键管理非常简单。但是,有些 API 可能存在性能问题。

首先,我们可能会发现,如果在 Redis 中存在大量的键,那么调用 KEYS 命令时会使 Redis 服务器阻塞一段时间,直到所有的键都返回完(在我们的示例中耗费了 6.8 秒)。如果理解了第 1 章开始使用 Redis 一章中理解 Redis 事件模型案例的核心概念,我们就会知道,在 Redis 中的一个命令执行过程期间,所有服务器接收到的其他命令都必须等待被处理。因此,对于生产环境的性能来说,调用 KEYS 命令是一个危险的操作。对于这个问题,可以使用此前案例中所提到的 SCAN 类命令,如 SCAN 或 SSCAN,以在不阻塞服务器的情况下在 Redis 服务器上遍历键。

此外,我们还应该对 DEL 命令额外留意。如果要删除的键是字符串以外的数据类型,那么当键中的元素数量很大时就可能会遭遇服务器延迟。为了避免这种灾难,应该使用 UNLINK 替代。UNLINK 会在另一个线程而不是主事件循环线程中执行删除操作,因而不会阻塞事件的处理。

此外,RENAME 命令乍一看似乎是无害的。但是,RENAME 会在目标键已存在时将其删除。如前所述,DEL 可能导致高延迟。因此,重命名操作的最佳实践是,如果目标键已存在则先对其执行 UNLINK,然后再进行重命名。

2.9.4 更多细节

DUMP/RESTORE 命令可以用于序列化和反序列化。我们可以使用这两个命令对 Redis 进行一部分的备份工作。

2.9.5 相关内容

- 请参阅<https://redis.io/commands#generic>获取完整的命令参考。

第 3 章

数据特性

在本章中，我们将学习下列案例：

- 使用位图（bitmap）。
- 设置键的过期时间。
- 使用 SORT 命令。
- 使用管道（pipeline）。
- 理解 Redis 事务。
- 使用发布订阅（PubSub）。
- 使用 Lua 脚本。
- 调试 Lua 脚本。

3.1 本章概要

我们在第 2 章数据类型中介绍了 Redis 支持的七种数据类型。除了这些在开发中常用的基本数据类型之外，Redis 还提供了一些有用的数据特性。如果我们能够学会如何正确地使用这些特性，那么我们的工作将变得更加轻松。

我们将在这一章讨论下列特性：

- **位图（Bitmap）**：在这个案例中，我们将展示在某些情况下如何使用位图代替字符串以节省内存空间。

- **过期时间 (Expiration)**：由于 Redis 是一个经常被用作缓存的内存数据存储，因此，对于临时数据，我们必须设置过期时间。在这个案例中，我们将展示如何为 Redis 中的键设置过期时间，以及键过期时发生的动作。
- **排序 (Sorting)**：Redis 支持对 Redis 的列表、集合、有序集合中的值进行排序并输出排序后的结果。我们将这个案例中展示 SORT 命令的用法。
- **管道 (Pipeline)**：在这个案例中，我们将学习 Redis 管道的使用，并了解管道之所以能够用于优化多个 Redis 操作性能的原因。
- **事务 (Transaction)**：Redis 支持类似于关系数据库的事务，但 Redis 的事务又与之不同。这个案例将对此进行详细说明。
- **发布订阅 (PubSub)**：Redis 可以被用作消息交换的通道 (message-exchanging channel)。我们会在这个案例中学习发布/订阅功能相关的命令及其应用场景。
- **编写/调试 Lua 脚本**：Lua 是一种被许多应用程序用作嵌入到程序内部的脚本语言，以此来实现程序的可配置性和可扩展性。在 Redis 中，Lua 脚本可以将多个操作打包在一起并原子性地执行。在这个案例中，我们将学习如何在 Redis 中编写、执行和调试简单的 Lua 脚本。

3.2 使用位图 (bitmap)

位图（也称为位数组或位向量）是由比特位 (bit) 组成的数组。Redis 中的位图并不是一种新的数据类型，它实际的底层数据类型是字符串。因为字符串本质上是二进制大对象 (BLOB, Binary Large Object)，所以可以将其视做位图。同时，因为位图存储的是布尔信息，所以在某些情况下可以节省大量的内存空间。

在本案例中，我们将使用位图来存储“用户是否曾经用过 Relp 中某个功能”的标志位。假设 Relp 中的每个用户都有一个惟一的递增 ID，那么用户 ID 就可以使用位图的偏移量表示。而“是否曾经用过 Relp 中某个功能”的标志位则是一个布尔属性，可以用位图中比特位的值来表示。

3.2.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

3.2.2 操作步骤

接下来，让我们按照以下的步骤来学习使用位图：

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。

2. 使用 `SETBIT` 命令设置位图指定偏移处比特位的值。

例如，当 `Relp` 中一个 ID 为 100 的用户使用过餐厅预订功能后，我们就可以设置相应的比特位：

```
127.0.0.1:6379> SETBIT "users_tried_reservation" 100 1
(integer) 0
```

3. 我们可以使用 `GETBIT` 命令，从位图中获取位于指定偏移处比特位的值。

例如，我们可以使用如下的命令判断用户 ID 为 400 的用户是否曾经使用过在线下单功能：

```
127.0.0.1:6379> GETBIT "users_tried_online_orders" 400
(integer) 0
```

4. 我们可以使用 `BITCOUNT` 命令获取位图中被设置为 1 的比特数。

例如，如果我们想计算有多少用户曾经使用过某个特定功能，可以使用 `BITCOUNT` 命令：

```
127.0.0.1:6379> BITCOUNT "users_tried_reservation"
(integer) 1
```

5. `BITOP` 命令用于进行位操作，该命令支持四种位操作：`AND`、`OR`、`XOR` 和 `NOT`。位运算的结果会被存储在一个目标键中。

例如，如果我们想获取曾经使用过餐厅预订功能和在线下单功能的用户 ID，那么可以：

```
127.0.0.1:6379> BITOP AND "users_tried_both_reservation_and_online_orders" "
users_tried_reservation" "users_tried_online_orders"
(integer) 13
127.0.0.1:6379> BITCOUNT "users_tried_both_reservation_and_online_orders"
(integer) 0
```

3.2.3 工作原理

Redis 中位图的结构如图 3.1 所示：

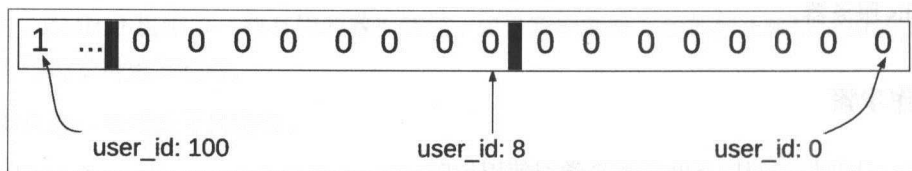


图 3.1 位图结构

3.2.4 更多细节

在前面的例子中,我们也可以使用集合来计算用户的数量。让我们从内存使用的角度,来看看在这个例子中使用位图和集合的区别。

正如我们所看到的,无论用户是否使用过某个功能,每个用户都需要占用位图中的一个比特。假设 Relp 有 20 亿个用户,那么我们就需要在内存中分配 20 亿个比特,也就是大约 250MB。如果我们使用集合来实现同样的计数功能,则只需在集合中存储使用过 Relp 特定功能的用户即可。

假定我们在 Relp 中使用 8 个字节的整型来存储用户 ID。如果 Relp 的某个功能非常流行且被 80% 的 Relp 用户(16 亿)使用过,那么我们需要为 16 亿个 8 字节的整型分配空间,也就是大约 12.8GB 内存。当集合中的元素非常多时,位图在节省存储空间方面具有优势。

不过,如果 Relp 的某个功能不那么流行,那么使用集合可能会更好。例如,如果只有 1% 的用户(2000 万)使用了某个功能,那么使用集合需要 160MB 的内存,而使用位图则仍然需要 250MB 的空间。我们暂且假定位图中比特位的分布是均匀分布的,在这种情况下,位图会是非常稀疏的。

此外,在一个稀疏的位图中设置比特位可能会将 Redis 服务器阻塞一段时间。当需要设置的比特位所在的偏移量很大而现有的位图又很小时,就会发生这种情况,因为 Redis 必须立即分配内存空间来容纳位图。

3.2.5 相关内容

- 有关 Redis 位图更详细的介绍和用法,请参阅<https://redis.io/topics/data-types-intro>中的位图一节。

3.3 设置键的过期时间

在第 2 章数据类型一章键管理的案例中,我们学习了使用 DEL 或 UNLINK 命令删除键。除了手动删除键之外,我们还可以通过设置键的超时时间让 Redis 自动地删除键。在本案例中,我们将演示如何在 Redis 中设置键的过期时间,并学习 Redis 中键过期的机制。

3.3.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器,并使用 redis-cli 连接到这个 Redis 服务器。

3.3.2 操作步骤

我们以“将距离用户当前位置最近的五个餐厅的 ID 存储到一个 Redis 列表中”为例，来展示如何设置键的过期时间。由于用户的位置可能经常改变，所以我们应该在这个列表上设置一个过期时间。在过期后，我们需要再次使用用户的当前位置获取餐厅列表。

1. 创建一个名为 `closest_restaurant_ids` 的餐厅 ID 的列表：

```
127.0.0.1:6379> LPUSH "closest_restaurant_ids"
109 200 233 543 222
(integer) 5
```

2. 使用 `EXPIRE` 命令将键的超时时间设置为 300 秒：

```
127.0.0.1:6379> EXPIRE "closest_restaurant_ids" 300
(integer) 1
```

3. 我们可以使用 `TTL` 命令，在键过期前查看剩余时间：

```
127.0.0.1:6379> TTL "closest_restaurant_ids"
(integer) 269
```

4. 等待 300 秒，直到键过期；之后，再使用 `EXISTS` 命令判断键是否存在时将返回 0。

```
127.0.0.1:6379> EXISTS "closest_restaurant_ids"
(integer) 0
```

3.3.3 工作原理

当对 Redis 中的一个键设置过期时间时，键的过期时间会被存储为一个绝对的 UNIX 时间戳。这样做的目的在于，即使 Redis 服务器宕机了一段时间，这个时间戳也会被持久化到 RDB 文件中（后面的章节将进行详细说明）。当 Redis 再次启动时，这个用来判断键是否过期的时间戳并不会发生变化，一旦当前时间超过了这个时间戳时，键就过期了。

在一个键过期后，当客户端试图访问已过期键时，Redis 会立即将其从内存中删除。Redis 这种删除键的方式被称为**被动过期（expiring passively）**。对于那些已经过期且永远不会再被访问到的键，Redis 还会定期地运行一个基于概率的算法来进行主动删除。更具体地说，Redis 会随机选择设置了过期时间的 20 个键。在这 20 个被选中的键中，已过期的键会被立即删除；如果选中的键中有超过 25% 的键已经过期且被删除，那么 Redis 会再次随机选择 20 个键并重复这个过程。默认情况下，上述过程每秒运行 10 次（可通过配置文件中 `hz` 的值进行设置）。

3.3.4 更多细节

我们可以通过以下方式清除一个键的过期时间：

- 使用 `PERSIST` 命令使其成为持久的键。
- 键的值被替换或删除。包括 `SET`、`GETSET` 和 `*STORE` 在内的命令会清除过期时间。不过，修改列表、集合或哈希的元素却不会清除过期时间，这是因为修改元素的操作并不会替换键所关联的值对象。
- 被另一个没有过期时间的键重命名。

如果键存在但未设置过期时间，则 `TTL` 命令返回 `-1`；如果键不存在，则返回 `-2`。

`EXPIREAT` 命令与 `EXPIRE` 命令类似，但它指定一个绝对 UNIX 时间戳为参数，该参数用于直接指定键的过期时间。

此外，从 Redis 2.6 开始，我们可以使用 `PEXIRE` 和 `PEXIREAT` 命令以毫秒级的精度指定键的过期时间。

因为 Redis 对已过期键的主动删除动作是不可预测的，所以有些已过期的键可能永远不会被删除。当发现有太多已过期的键没有被删掉时，我们可以通过执行 `SCAN` 命令来更加“主动地”触发被动过期。

3.3.5 相关内容

- `EXPIRE` 命令的官方文档，请参阅：<https://redis.io/commands/expire>。

3.4 使用 SORT 命令

我们已经在第 2 章数据类型一章中了解到，Redis 列表或集合中的元素是无序的，而有序集合中的元素是根据其权重排序的。有时，我们可能需要获取一个 Redis 列表或集合的已排序副本，或者以某种非权重顺序对有序集合中的元素进行排序。Redis 为这些需求提供了一个方便的命令 `SORT`。在本案例中，我们将学习 `SORT` 命令及其示例。

3.4.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

3.4.2 操作步骤

接下来，让我们按照以下的步骤来学习 SORT 命令的使用：

1. 打开一个终端，并使用 `redis-cli` 连接到 Redis。
2. 如果所有的元素都是数值，那么我们可以简单地运行 SORT 命令按升序对元素排序。假设在 Relp 中我们将用户喜爱餐厅的 ID 存储在 Redis 集合中，那么可以使用：

```
127.0.0.1:6379> SADD "user:123:favorite_restaurant_ids" 200 365 104 455 333
(integer) 5
127.0.0.1:6379> SORT "user:123:favorite_restaurant_ids"
1) "104"
2) "200"
3) "333"
4) "365"
5) "455"
```

3. 如果存在非数值的元素且想按字典顺序对它们进行排序，那么我们需要增加修饰符 ALPHA。假设我们把餐厅的名称存储在集合中并希望按字典顺序排序：

```
127.0.0.1:6379> SADD "user:123:favorite_restaurants" "Dunkin Donuts" "Subway"
"KFC" "Burger King" "Wendy's"
(integer) 5
127.0.0.1:6379> SORT "user:123:favorite_restaurants" ALPHA
1) "Burger King"
2) "Dunkin Donuts"
3) "KFC"
4) "Subway"
5) "Wendy's"
```

4. 在使用 SORT 命令时添加修饰符 DESC 会按降序返回元素。默认情况下，SORT 命令会排序并返回所有元素；但是，我们可以通过使用 LIMIT 修饰符来限制返回元素的数量。在使用 LIMIT 修饰符时，我们需要同时指定起始偏移量（要跳过元素的数量）和数量（要返回元素的数量）。例如，如果我们只需要找出用户最喜欢的前三家餐厅（按字典顺序排序）：

```
127.0.0.1:6379> SORT "user:123:favorite_restaurants" ALPHA LIMIT 0 3
1) "Burger King"
2) "Dunkin Donuts"
3) "KFC"
```


3.4.3 更多细节

有时, 我们不希望按值对元素进行排序, 而是按在某些其他键中定义的权重来对元素进行排序。举例来说, 我们可能需要按照定义在形如 `restaurant_rating_200` (其中的 200 是餐厅的 ID) 的键中的评级对用户喜欢的餐厅进行排序。这也可以使用 `SORT` 完成:

```
127.0.0.1:6379> SET "restaurant_rating_200" 4.3
127.0.0.1:6379> SET "restaurant_rating_365" 4.0
127.0.0.1:6379> SET "restaurant_rating_104" 4.8
127.0.0.1:6379> SET "restaurant_rating_455" 4.7
127.0.0.1:6379> SET "restaurant_rating_333" 4.6
127.0.0.1:6379> SORT "user:123:favorite_restaurant_ids" BY restaurant_rating_*
DESC
1) "104"
2) "455"
3) "333"
4) "200"
5) "365"
```

在某些情况下, 外部键中的值更为有用。以之前的示例为例, 我们更感兴趣的是获得餐厅的名称而不是餐厅的 ID。假设我们将餐厅的名称存储在形如 `restaurant_name_200` (其中的 200 是餐厅的 ID) 的键中, 那么可以使用 `SORT` 命令的 `GET` 选项来获取餐厅的名称:

```
127.0.0.1:6379> SET "restaurant_name_200" "Ruby Tuesday"
127.0.0.1:6379> SET "restaurant_name_365" "TGI Friday"
127.0.0.1:6379> SET "restaurant_name_104" "Applebee's"
127.0.0.1:6379> SET "restaurant_name_455" "Red Lobster"
127.0.0.1:6379> SET "restaurant_name_333" "Boiling Crab"
127.0.0.1:6379> SORT "user:123:favorite_restaurant_ids" BY restaurant_rating_*
DESC GET restaurant_name_*
1) "Applebee's"
2) "Red Lobster"
3) "Boiling Crab"
4) "Ruby Tuesday"
5) "TGI Friday"
```

`GET` 选项可被多次使用, `GET #` 表示获取元素本身。

`SORT` 命令还有一个 `STORE` 选项, 该选项会把排序的结果当做列表保存到指定的键中。下面是一个使用 `STORE` 选项的示例:

```
127.0.0.1:6379>SORT "user:123:favorite_restaurant_ids" BY restaurant_rating_*
DESC GET restaurant_name_* STORE user:123:favorite_restaurant_names:
sort_by_rating
(integer) 5
```

最后，SORT 命令的时间复杂度是 $O(N+M*\log(M))$ ，其中 N 是列表或集合中元素的个数，而 M 是要返回的元素的数目。鉴于 SORT 操作的时间复杂度，在对大量数据进行排序时，Redis 服务器的性能可能会降低，这一点需要格外留心。

3.4.4 相关内容

- SORT 的官方文档，请参阅：<https://redis.io/commands/sort>。

3.5 使用管道（pipeline）

在第 1 章开始使用 Redis 一章理解 Redis 通信协议的案例中，我们已经了解到 Redis 客户端和服务端是通过 RESP 协议进行通信的。客户端和服务端之间典型的通信过程可以看作：

1. 客户端向服务器发送一个命令。
2. 服务器接收该命令并将其放入执行队列（因为 Redis 是单线程的执行模型）。
3. 命令被执行。
4. 服务器将命令执行的结果返回给客户端。

上述过程耗费的所有时间称为往返时延（RTT, round-trip time）。不难发现，第 2 步和第 3 步耗费的时间取决于 Redis 服务器，而第 1 步和第 4 步耗费的时间则完全取决于客户端和服务端之间的网络延迟。如果我们需要执行多个命令，那么与服务器执行命令耗费的时间（通常非常短）相比，网络传输可能会花费大量的时间。

使用 Redis 管道可以加快上述的过程。Redis 管道的基本思想是，客户端将多个命令打包在一起，并将它们一次性发送，而不再等待每个单独命令的执行结果；同时，Redis 管道需要服务器在执行所有的命令后再返回结果。即便是执行多个命令，但由于第 1 步和第 4 步只发生一次，所以总的执行时间会大大减少。

我们将在本案例中演示一个使用管道的简单示例。

3.5.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

使用以下的命令在 Ubuntu 中安装 dos2unix 工具：

```
sudo apt-get install dos2unix
```

在 macOS 中，我们可以使用以下的命令安装 dos2unix 工具：

```
brew install dos2unix
```

3.5.2 操作步骤

接下来，让我们按照以下的步骤来学习 pipeline 的使用：

1. 打开一个终端，然后执行以下的命令：

```
~$ cat pipeline.txt
set mykey myvalue
sadd myset value1 value2
get mykey
scard myset
```

这个文本文件中的每一行都必须以 `\r\n`，而不是 `\n` 结束。我们可以使用命令 `unix2dos` 实现：

```
$ unix2dos pipeline.txt
```

2. 使用 `redis-cli` 的 `--pipe` 选项，通过管道发送命令：

```
~$ cat pipeline.txt | bin/redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 4
```

3.5.3 工作原理

`redis-cli` 中的 `--pipe` 选项会一次性地发送所有来自 `stdin` 的命令，从而极大地减少往返时延的开销。

此外，我们也可以通过构造客户端与 Redis 服务器通信所用的原始 RESP 协议报文来发送命令。如前例，以下的操作可以实现相同的效果：

```
$ cat datapipe.txt
```



```
*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n*4\r\n$4\r\nSADD\r\n$5\r\n
  myset\r\n$6\r\nvalue1\r\n$6\r\nvalue2\r\n*2\r\n$3\r\nGET\r\n$5\r\nmykey\r\n
  \r\n*2\r\n$5\r\nSCARD\r\n$5\r\nmyset\r\n\r\n
$ echo -e "$(cat datapipe.txt)" | bin/redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 4
```

另外，通过 `redis-cli` 的 `--pipe` 选项，我们可以用原始 RESP 协议把所有命令一起发送到服务器。对于 RESP 协议的解释，请参阅第 1 章开始使用 *Redis* 一章理解 *Redis* 通信协议一节。

3.5.4 更多细节

我们演示了如何使用原始 RESP 协议将多个命令通过管道发送给 *Redis* 服务器。管道在特定编程语言的 *Redis* 客户端中经常被用到。我们将在第 4 章使用 *Redis* 进行开发中学习更多的相关细节。

3.5.5 相关内容

- 请参阅<https://redis.io/topics/pipelining>学习官方 *Redis* 文档中有关管道的详尽说明。

3.6 理解 *Redis* 事务 (transaction)

关系数据库中的事务是一组需要原子化执行的操作，这意味着一组操作必须同时成功或失败。但是在 *Redis* 中，事务的概念完全是另外一回事。在本案例中，我们将学习 *Redis* 事务，以及它与关系数据库事务之间的区别。

3.6.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 *Redis* 服务器，并使用 `redis-cli` 连接到这个 *Redis* 服务器。

3.6.2 操作步骤

为了学习 *Redis* 的事务，让我们为 *Relp* 中的某个餐厅组织一场在线快销秒杀优惠券的活动。这场秒杀活动只提供 5 张优惠券，并使用键 `counts:seckillling` 作为计数器来保存可用优惠券的数量。

下面是实现这个计数器的伪代码：

```
//初始化优惠券的数量
SET("counts:seckill",5);

Start decreasing the counter:
WATCH("counts:seckill");
count = GET("counts:seckill");
MULTI();
if count > 0 then
    DECR("counts:seckill",1);
    EXEC();
    if ret != null then
        print "Succeed!"
    else
        print "Failed!"
else
    DISCARD();
    print "Seckill Over!"
```

3.6.3 工作原理

对于秒杀类应用的设计而言，超卖经常是一个令人头痛的难题。超卖发生的原因在于，当我们获取计数器值的时候涉及竞态；计数器的值也许对业务场景而言是有效的（译者注：即大于零），但是在减少计数器的值之前，计数器的值可能已经被其他请求修改了。在这个例子中，我们可以使用 Redis 的事务来避免这种情况发生。

首先，我们对一个键使用 WATCH 命令来设置一个标志，该标志用来判断在执行 EXEC 命令之前是否修改了键；如果修改了键，那么就丢弃整个事务。然后，获取计数器的值。

接下来，我们通过调用 MULTI 命令来启动一个事务。如果计数器的值无效（译者注：即小于等于零），则使用 DISCARD 命令直接放弃该事务。否则，继续减少计数器的值。

在此之后，我们尝试执行事务。由于之前使用过 WATCH 命令，所以 Redis 会检查计数器 counts:seckill 的值是否已被修改。如果值被修改过，则中止事务。这种中止就被视作秒杀失败。

如读者所见，我们通过利用 Redis 中的事务成功地避免了超卖的错误。

3.6.4 更多细节

我们需要注意关系数据库事务和 Redis 事务之间的区别。

它们之间的关键区别在于，Redis 事务没有回滚功能。一般来说，在一个 Redis 事务中可能会出现两种类型的错误，而针对这两种类型的错误会采取不同的处理方式：

1. 第一种错误是命令有语法错误。在这种情况下，由于在命令入队时就能发现存在语法错误，所以整个事务会快速失败且事务中的所有命令都不会被处理。

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET FOO BAR
QUEUED
127.0.0.1:6379> GOT FOO
(error) ERR unknown command 'GOT'
127.0.0.1:6379> INCR MAS
QUEUED
127.0.0.1:6379> EXEC
(error) EXECABORT Transaction discarded because of previous errors.
```

2. 第二种错误是，虽然所有命令都已成功入队，但在执行过程中发生了错误。位于发生错误命令之后的其他命令将继续执行，而不会回滚。以下面的事务为例：

```
127.0.0.1:6379> MULTI
OK
127.0.0.1:6379> SET foo bar
QUEUED
127.0.0.1:6379> INCR foo
QUEUED
127.0.0.1:6379> SET foo mas
QUEUED
127.0.0.1:6379> GET foo
QUEUED
127.0.0.1:6379> EXEC
1) OK
2) (error) ERR value is not an integer or out of range
3) OK
4) "mas"
```

3.6.5 相关内容

- 有关 Redis 事务的详细讨论，请参阅：<https://redis.io/topics/transactions>。

- 对于事务命令，请参阅：<https://redis.io/commands#transactions>。

3.7 使用发布订阅（PubSub）

发布-订阅（**Publish-Subscribe, PubSub**）是一种历史悠久的经典消息传递模式。根据维基百科的说法，发布-订阅模式早在 1987 年就出现了。简单地说，在发布-订阅模式中，想要发布事件（event）的发布者（publisher）会把消息（message）发送到一个 PubSub 频道（channel），这个频道会把事件投递（deliver）给对这个频道感兴趣的每一个订阅者（subscriber）。许多流行的消息传递中间件，比如 Kafka 和 ZeroMQ，就利用了这个模式来构建消息投递系统；Redis 也是如此。在本案例中，我们将学习 Redis 的 PubSub 功能。

3.7.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

3.7.2 操作步骤

为了展示如何使用 PubSub，让我们举一个推荐消息推送系统（recommendation message-pushing system）的例子。

打开三个控制台来模拟两个订阅者和一个发布者，订阅者为 console-A（SUBer-1）和 console-B（SUBer-2）、发布者为 console-C（PUBer）：

1. 在 SUBer1 中订阅 “restaurants:Chinese” 频道：

```
127.0.0.1:6379> SUBSCRIBE restaurants:Chinese
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "restaurants:Chinese"
3) (integer) 1
```

2. 在 SUBer2 中订阅 “restaurants:Chinese” 和 “restaurants:Thai” 频道：

```
127.0.0.1:6379> SUBSCRIBE restaurants:Chinese restaurants:Thai
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "restaurants:Chinese"
3) (integer) 1
```

- 1) "subscribe"
- 2) "restaurants:Thai"
- 3) (integer) 2

3. 在 PUBer 中向 “restaurants:Chinese” 频道发布消息：

```
127.0.0.1:6379> PUBLISH restaurants:Chinese "Beijing roast duck discount
tomorrow"
(integer) 2
```

4. 两个订阅者将收到以下的信息：

- 1) "message"
- 2) "restaurants:Chinese"
- 3) "Beijing roast duck discount tomorrow"

5. 在 PUBer 中向 “restaurants:Thai” 频道发布消息：

```
127.0.0.1:6379> PUBLISH restaurants:Thai "3$ for Tom yum soap in this weekend
!"
(integer) 1
```

6. 只有订阅了 “restaurants:Thai” 频道的 SUBer-2 才会收到这条信息：

- 1) "message"
- 2) "restaurants:Thai"
- 3) "3\$ for Tom yum soap in this weekend!"

3.7.3 工作原理

SUBSCRIBE 命令用来监听特定频道中的可用消息。一个客户端可以使用 SUBSCRIBE 命令一次性地订阅多个频道，也可以使用 PSUBSCRIBE 命令订阅匹配指定模式的频道。要取消订阅频道，可以使用 UNSUBSCRIBE 命令。

PUBLISH 命令用于将一条消息发送到指定的频道。订阅了该频道的所有订阅者将接收到这条消息。

另一个重要的命令是 PUBSUB，用于进行频道管理。例如，我们可以通过 PUBSUB CHANNELS 命令获取当前活跃的频道，如下所示：

```
127.0.0.1:6379> PUBSUB CHANNELS
```

```
1) "restaurants:Chinese"
```

```
2) "restaurants:Thai"
```

3.7.4 更多细节

对频道的生命周期而言，如果给定的频道之前未曾被订阅过，那么 `SUBSCRIBE` 命令会自动创建频道。此外，当频道上没有活跃的订阅者时，频道将会被删除。

读者需要务必注意，PubSub 相关的机制均不支持持久化。这意味着，消息、频道和 PubSub 的关系均不能保存到磁盘上。如果服务器出于某种原因退出，那么所有的这些对象都将丢失。

此外，在消息投递和处理场景中，如果频道没有订阅者，那么被发到频道上的消息将被丢弃。换句话说，Redis 并没有保证消息投递可靠性的机制。

总之，虽然 Redis 中的 PubSub 功能并不适合重要消息的投递场景，但是有些人可能会由于其简洁的通信方式而在速度方面获益。

此外，Redis 中基于 PubSub 的键空间通知（`keyspace notification`）功能允许客户端订阅一个 Redis 频道来接收命令发布或数据改变的事件。如果读者对这个功能感兴趣，可以参阅本节相关内容中列出的文档。

3.7.5 相关内容

- 有关 PubSub 的详细讨论，请参阅：<https://redis.io/topics/pubsub>。
- 有关 PubSub 相关命令的细节，请参阅：<https://redis.io/commands#pubsub>。
- 有关键空间通知功能，请参阅：<https://redis.io/topics/notifications>。

3.8 使用 Lua 脚本

Lua 是一种轻量级的脚本语言，Redis 从 2.6 版本开始引入对 Lua 脚本的支持。与在理解 Redis 事务一节中提到的 Redis 事务类似，Lua 脚本是原子化执行的；不过，Lua 语言作为服务器端脚本语言，能够实现更为强大的功能和程序逻辑。在本案例中，我们将学习如何在 Redis 中编写和执行 Lua 脚本。

3.8.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

3.8.2 操作步骤

接下来，让我们按照以下的步骤来学习使用 Lua 脚本：

1. 我们将使用 Lua 脚本来更新 Redis 中的一个 JSON 字符串对象。
2. 打开一个控制台并创建一个 Lua 脚本，如下：

```
$ mkdir /redis/coding/lua; cd /redis/coding/lua
$ cat updatejson.lua
local id = KEYS[1]
local data = ARGV[1]
local dataSource = cJSON.decode(data)

local retJson = redis.call('get', id)
if retJson == false then
    retJson = {}
else
    retJson = cJSON.decode(retJson)
end

for k,v in pairs(dataSource) do
    retJson[k] = v
end

redis.call('set', id, cJSON.encode(retJson))
return redis.call('get',id)
```

3. 我们可以使用 `redis-cli` 来执行 Lua 脚本：

```
bin/redis-cli --eval updatejson.lua users:id:992452 , '{"name": "Tina", "sex":
    "female", "grade": "A"}'
{"grade": "A", "name": "Tina", "sex": "female"}
```

4. 如果要在后续调用该脚本，那么可以将其注册到 Redis 服务器中：

```
bin/redis-cli SCRIPT LOAD "`cat updatejson.lua`"
"45a40b129ea0655db7e7be992f344468559f3dbd"
```

5. 在脚本注册之后，我们可以通过指定注册脚本时返回的唯一 SHA-1 标识符来执行这个 Lua 脚本。在这里，我们把用户 `users:id:992452` 的等级更新为 “c”：

```
bin/redis-cli EVALSHA 45a40b129ea0655db7e7be992f344468559f3dbd 1 users:id
:992452 '{"grade": "C"}'
{"grade": "C", "name": "tina", "sex": "female"}
```

3.8.3 工作原理

首先，我们创建了一个名为 `updatejson.lua` 的 Lua 脚本。在这个 Lua 脚本中，KEY 和 ARGV 是 EVAL 命令的参数。在脚本的开头，我们获取 KEY 作为要处理的键、ARGV 作为要更新的 JSON 字符串的内容。在这之后，将传递给脚本的 JSON 内容反序列化。

然后，我们通过使用 `redis.call()` 函数调用 GET 命令获取指定键的值，随后查看 GET 命令返回的值。如果值为 `false`，则表示键不存在，我们就将其设置为空表。否则，则将这个值作为一个 JSON 字符串进行解析。

随后，我们开始遍历数据（译者注：即传入参数中 JSON 字符串反序列化后所得到表的字段）来设置 `retJson` 变量中的键和值。在这个脚本的最后，我们把 JSON 字符串设置成键值，并返回最终结果。

我们可以使用 `redis-cli` 的 `--eval` 选项来执行一个 Lua 脚本。在本例中，我们把要更新的用户 ID 和用户信息传给了 `redis-cli`：

```
users:id:992452 , '{"name": "Tina", "sex": "female", "grade": "A"}'
```

在这个 Lua 脚本中，用户 ID 被当做 `KEYS[]` 的元素，用户信息被当做 `ARGV[]` 的元素；逗号将 `KEYS[]` 与 `ARGV[]` 分隔开。在 Lua 脚本中，`KEYS[]` 数组和 `ARGV[]` 数组的索引都是从 1 开始的。除了使用 `redis-cli` 外，我们还可以在程序中使用 EVAL 命令来执行 Lua 脚本。

实际上，我们并不需要在每次执行时都指定 Lua 脚本。在这种情况下，`SCRIPT LOAD` 命令可以将脚本在 Redis 服务器进程中缓存起来，并返回一个 SHA-1 字符串作为这个脚本的唯一标识。当我们想执行这个脚本时，把该 SHA-1 字符串作为 EVALSHA 命令的参数传入即可。

3.8.4 更多细节

因为 Redis 中的 Lua 脚本和事务都可以将一组操作原子化地执行，所以读者可能想了解如何在它们之间进行选择。一般来说，当我们的业务场景中涉及复杂逻辑判断或循环处理时，最好选择 Lua 脚本而不是事务。

与 Redis 事务类似，我们也必须注意 Lua 脚本的执行时间。在执行 Lua 脚本期间，Redis 服务器不能处理任何其他命令。因此，我们必须确保 Lua 脚本可以尽快地执行完。在默认情况下，一旦 Lua 脚本的执行时间超过了 Lua 脚本运行时长限制（默认为 5 秒，在 Redis 配置文件中的 `lua-time-limit`

选项可以修改默认值)，我们就可以调用 `SCRIPT KILL` 来将其终止。但是，如果此正在运行的脚本中已经调用了任何写入相关的命令，那么我们就不得不采用无持久化的 `SHUTDOWN NOSAVE` 命令来关闭 Redis 服务器以中止此脚本的运行。如果某个 Lua 脚本的执行时间小于 Lua 脚本运行时长限制，`SCRIPT KILL` 命令则会被阻塞，直到脚本执行超时。

对于 Lua 脚本的管理而言，我们可以通过调用 `SCRIPT EXISTS` 命令并传入脚本的 SHA-1 标识来判断一个脚本是否已存在并注册。

最后要注意的一点是，因为脚本只是被保存在 Redis 服务器进程的脚本缓存中，而脚本缓存在重新启动时将会消失，所以在重新启动 Redis 服务器之后必须重新加载 Lua 脚本。

3.8.5 相关内容

- Lua 5.1 的完整手册，请参阅：<https://www.lua.org/manual/5.1/>。
- 关于 Redis 的 Lua 命令，请参阅：<https://redis.io/commands#scripting>。

3.9 调试 Lua 脚本

调试使得我们能够发现、诊断和消除程序中的错误。一般来说，调试一个程序涉及检查逻辑和查看变量的值。对于 Lua 脚本而言，Redis 从 3.2 版本开始引入了一个调试工具来简化调试过程，并提供了一些函数来帮助我们打印调试日志。在本案例中，我们将学习在 Redis 中调试 Lua 脚本的步骤。

3.9.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

要在 Redis 中打印日志，我们需要在 `redis.conf` 中配置以下选项：

```
logfile "/redis/log/redis.log"
loglevel debug
```

然后，按照第 1 章开始使用 Redis 一章中启动和停止 Redis 案例所展示的步骤，使用修改后的配置文件启动 Redis 服务器。

3.9.2 操作步骤

接下来，让我们以上一节中使用的 Lua 脚本为例，演示如何调试 Redis 的 Lua 脚本：

1. 打开一个控制台并修改脚本，如下：

```
$ mkdir /redis/coding/lua; cd /redis/coding/lua
$ cat updatejsondebug.lua
local id = KEYS[1]
local data = ARGV[1]
redis.debug(id)
redis.debug(data)

local retJson = redis.call('get', id)
local dataSource = cJSON.decode(data)

if retJson == false then
    retJson = {}
else
    retJson = cJSON.decode(retJson)
end
redis.breakpoint()

for k,v in pairs(dataSource) do
    retJson[k] = v
end

redis.log(redis.LOG_WARNING,cJSON.encode(retJson))
redis.call('set', id, cJSON.encode(retJson))

return redis.call('get',id)
```

2. 通过 redis-cli 设置用户 ID 为 992398 的用户属性：

```
$ bin/redis-cli set users:id:992398 '{"grade":"C","name":"Mike","sex":"male"}'
```

3. 为了验证我们是否能够通过该 Lua 脚本达到修改 JSON 的结构来记录用户级别变化这一目的，使用 redis-cli 的 --ldb 选项启动一次程序调试。

```
$ bin/redis-cli --ldb --eval updatejsondebug.lua users:id:992398 , '{"grade":
{"init":"C","now":"A"}}'
Lua debugging session started, please use:
quit -- End the session.
```

```
restart -- Restart the script in debug mode again.  
help    -- Show Lua script debugging commands.
```

```
* Stopped at 1, stop reason = step over  
-> 1    local id = KEYS[1]
```

4. 键入 `s` 来单步地执行脚本，调试信息将被 `redis.debug()` 函数打印出来：

```
lua debugger> s  
* Stopped at 3, stop reason = step over  
-> 3    redis.debug(id)  
lua debugger> s  
<debug> line 3: "users:id:992398"  
* Stopped at 4, stop reason = step over  
-> 4    redis.debug(data)  
lua debugger> s  
<debug> line 4: "{"grade": {"init":"C","now":"A"}}"  
* Stopped at 6, stop reason = step over  
-> 6    local retJson = redis.call('get', id)
```

5. 键入 `s` 来继续执行脚本。

当 Redis 命令在 Lua 脚本中被调用时，调试器会显示出完整的命令及其执行结果。我们可以使用 `p VARIABLE` 输出 Lua 脚本中变量的值：

```
lua debugger> s  
<redis> get users:id:993298  
<reply> "{"grade":"C","name":"Mike","sex":"male"}"  
* Stopped at 7, stop reason = step over  
-> 7    local dataSource = cJSON.decode(data)  
lua debugger> p retJson  
<value> "{"grade":"C","name":"Mike","sex":"male"}"
```

6. 然后，继续执行脚本。脚本会停在我们使用 `redis.breakpoint()` 所设置断点的下一个有效表达式：

```
lua debugger> c  
* Stopped at 16, stop reason = redis.breakpoint() called  
-> 16    for k,v in pairs(dataSource) do
```

7. 键入 `w` 可以浏览我们正在调试的整个脚本。在浏览完整个脚本后,我们可以使用 `b LINENUMBER` 设置一个断点,以便在将修改后的 JSON 设置到键中前检查结果。之后,使用 `c` 继续执行,直到运行到上一步设置的断点:

```
lua debugger> b 21
20 redis.log(redis.LOG_WARNING,cjson.encode(retJson))
#21 redis.call('set', id, cjson.encode(retJson))
22
lua debugger> c
* Stopped at 21, stop reason = break point
->#21 redis.call('set', id, cjson.encode(retJson))
```

函数 `redis.log()` 会把相应的日志信息写入到 Redis 服务器的日志文件中,在本例中就是位于 `/redis/log` 目录下的文件 `redis.log`:

```
15549:M 26 Sep 09:20:52.442 # {"grade":"C","name":"Mike","sex":"male"}
```

8. 接下来,继续执行脚本直到结束:

```
lua debugger> c
{"grade":{"now":"A","init":"C"},"name":"Mike","sex":"male"}"
(Lua debugging session ended -- dataset changes rolled back)
```

3.9.3 工作原理

上述的调试过程很容易理解,而且我们也不需要记住 `ldb` 中的所有命令;直接通过 `ldb` 的 `help` 命令获取帮助信息即可。

很重要的一点是,因为调试会话创建了单独的进程,所以 Redis 服务器在调试过程中不会被阻塞。此外,这种机制意味着,在调试模式下做出的所有更改在默认情况下都将被回滚。另一个选项 `--ldb-sync-mode` 则会使服务器在调试期间不可用。所以,在确实有必要时,我们可以将在调试模式设置同步方式。

我们将在第 8 章生产环境部署一章中日志的案例中详细学习 Redis 日志相关的内容。

3.9.4 更多细节

当我们想查看 Lua 脚本中某个变量的值时,可能首先想到的是直接用 Lua 语言中的 `print()` 函数来输出变量值。虽然这样做很直接了当且确实可行,但我们必须注意到,如果服务器是在后台

服务模式下运行的，那么就看不到日志。这是因为，`print()` 生成的消息会显示在 `redis-server` 进程的 `stdout` 中，而如果服务器是以后台服务模式启动的，`stdout` 会被丢弃。

3.9.5 相关内容

- 有关 **Redis Lua 调试器（LDB）** 的完整手册，请参阅：<https://redis.io/topics/ldb>。
- 有关 Redis Lua 脚本的调试命令，请参阅：<https://redis.io/commands/script-debug>。
- 如果读者对更多的调试技术感兴趣，请参阅：<https://redislabs.com/blog/5-6-7-methods-for-tracing-and-debugging-redis-lua-scripts/>。

第 4 章

使用 Redis 进行开发

在本章中，我们将学习下列案例：

- Redis 常见应用场景。
- 使用正确的数据类型。
- 使用正确的 Redis API。
- 使用 Java 连接到 Redis。
- 使用 Python 连接到 Redis。
- 使用 Spring Data 连接到 Redis。
- 使用 Redis 编写 MapReduce 作业。
- 使用 Redis 编写 Spark 作业。

4.1 本章概要

我们在第 2 章数据类型和第 3 章数据特性中学习了 Redis 的数据类型和许多有用的特性。在本章中，我们将重点讨论使用 Redis 进行应用开发的主题。

首先，我们将介绍一些常见的 Redis 用例，让读者大致了解在现实世界中 Redis 是如何工作的。

在此之后，我们为如何选择正确的数据类型和 API 提供一些指导原则。在使用 Redis 进行应用开发时，遵循这些指导原则是非常必要的。这是因为，Redis 与大多数关系数据库不同，除了微调一些配置参数以增强 Redis 的处理能力以外，我们几乎在 Redis 服务器端不能进行任何其他优化。在应用程序设计的最开始，使用恰当的数据类型和 API，是充分利用 Redis 的高性能特性并同时避免其短处的关键。

在学习了设计指导原则后，我们将展示如何在 Java 和 Python 开发的应用程序中使用 Redis。随后，针对 Web 应用程序开发，我们讲解了如何在 Spring Data 中使用 Redis。

最后，本章的最后两个案例会介绍如何在大数据的世界中使用 Redis。

4.2 Redis 常见应用场景

我们在此前的章节中已经学习了 Redis 的强大功能，但读者可能还想知道 Redis 能在一个应用程序中完成什么样的工作，以及应该在什么情况下使用 Redis。在本案例中，我们将展示一些实际的应用场景。在这些场景中，Redis 是较其他存储解决方案更优的选择。通过对本案例的学习，我们希望读者能收获更多有关在应用程序中使用 Redis 的思考。

4.2.1 会话存储

在现代网站的架构中，通常多个 Web 服务器位于一个或多个负载均衡器之后。会话（Session）通常需要存储在外部分布式存储系统中。如果有一个 Web 服务器宕机，其他的服务器可以从外部存储中获取会话并继续服务。因为与关系数据库相比 Redis 的访问延迟非常低，所以使用 Redis 来保存会话数据堪称是一种完美的会话存储机制。此外，Redis 中对键过期的支持可以天然地用于会话的超时管理。

4.2.2 分析

Redis 还可以用于分析和统计的场景。例如，如果我们想要计算有多少用户查看了一个餐厅，那么简单地使用 INCR 命令增加统计餐厅被查看数的计数器即可。因为所有的 Redis 命令都是原子的，所以我们无需担心竞态。基于诸如哈希、有序集合和 HyperLoglog 等数据类型，我们还可以构建其他更高级的计数器或统计数据捕获系统。

4.2.3 排行榜

在 Redis 的有序集合的帮助下，我们可以轻松地实现一个排行榜。正如在第 2 章数据类型一章中使用有集合（Sorted Set）类型的案例所展示的那样，我们可以为餐厅创建一个有序集合并将用户的投票数作为权重。因此，使用 ZREVRANGE 命令就能够按照餐厅的受欢迎程度返回餐厅的列表。同样的功能也可以在诸如 MySQL 等的关系数据库中实现，但是 SQL 查询要比 Redis 查询慢得多。

4.2.4 队列

我们在第 2 章数据类型一章中使用列表（list）类型的案例中介绍了列表的 PUSH/POP 命令（阻塞类型）。正如我们在那个案例中所演示的，Redis 的列表可以用来实现简单的任务队列。被广泛

使用的 **Resque** 项目（一个以 Redis 作为后端、用于任务排队的 Ruby 库）正是基于这个想法。通过 **RPOPLPUSH** 命令，我们还可以使用 Redis 列表实现可靠的队列（译者注：此处的可靠是相对的）。

4.2.5 最新的 N 个记录

假设我们想获得最近被添加到应用中的 10 家餐厅。如果使用关系数据库，那么我们需要运行一个 SQL 查询，例如：

```
SELECT * FROM restaurants ORDER BY created_at DESC LIMIT 10
```

我们可以使用 Redis 列表来解决这个问题：

1. 维护一个 Redis 列表，名为 `latest_restaurants`。
2. 在增加了新餐厅后，执行 `LPUSH latest_restaurants` 和 `LTRIM recent_restaurants 0 10` 命令。通过这种方式，Redis 的列表将始终包含最新增加的 10 家餐厅。

4.2.6 缓存

由于 Redis 是一种基于内存的数据存储系统，在关系数据库前面使用 Redis 作为缓存通常能够加速数据库的查询过程。在这里，让我们举一个简单的示例：在查询关系数据库之前，我们首先在 Redis 中查找记录。如果在 Redis 中找不到记录，则查询关系数据库并将记录放到 Redis 中。在向关系数据库写入时，我们也将记录写入 Redis。为了限制缓存的大小，可以对缓存中的记录设置过期时间或应用诸如最近最少使用（**Least Recently Used, LRU**）的回收策略。

4.2.7 更多细节

正如之前一些用例中所展示的，某些在关系数据库中运行缓慢或难以完成的任务在 Redis 中可以以很快速和轻松地完成。但是，Redis 并不能满足所有的存储需求。首先，因为 Redis 默认将所有数据存储在内存中（虽然一些基于云的 Redis 服务提供了使用 SSD 作为数据存储后端的选项），所以数据大小超过内存大小时 Redis 将无法容纳所有数据。其次，Redis 事务并不完全符合 ACID 规范（原子性、一致性、隔离性和持久性）。如果需要完全符合 ACID 规范的事务，就不能使用 Redis。在这些场景中，应该使用关系数据库或其他数据库系统。

4.2.8 相关内容

- 有关 ACID 的意义，请参阅：<https://en.wikipedia.org/wiki/ACID>。
- 对于 Resque 项目，请参阅：<https://github.com/resque/resque>。

4.3 使用正确的数据类型

在第 2 章数据类型中，我们学习了 Redis 为满足业务需求所提供的丰富数据类型。当我们开发一个使用 Redis 的应用程序时，先熟悉这些数据类型是非常重要的。我们不仅要理解它们之间的语法差异，还要分辨它们在不同业务场景中的优缺点。虽然在设计应用程序时，选出满足我们需求的一种或多种数据类型并不难，但通常在考虑到性能和内存消耗时总会有进一步优化空间。

在本案例中，我们将通过设计一个用户数据存储的示例来展示如何使用正确的数据类型来减少内存的消耗。

4.3.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

在学习下一节的内容之前，我们需要使用 `FLUSHALL` 命令清除 Redis 实例中的所有数据。

在 Ubuntu 中，我们可以使用以下的命令安装 `dos2unix` 工具：

```
sudo apt-get install dos2unix
```

在 macOS 中，我们可以使用以下的命令安装 `dos2unix` 工具：

```
brew install dos2unix
```

4.3.2 操作步骤

为了说明如何选择正确的数据类型，假定我们希望在 Redis 中保存 Relp 的用户信息。为了验证我们的原型设计是否正确，我们会将 10,000 个用户的信息导入到 Redis 中作为示例数据，以检查在 Redis 中存储用户信息时是否使用了最佳的策略。这意味着在满足应用程序的业务需求的同时，我们还会尽可能少地使用内存。出于演示的考虑，我们存储一个用户的下列信息：

- `id`：用户 ID。
- `name`：用户姓名。
- `sex`：性别（男/女）。
- `register_time`：注册时间。
- `nation`：用户国籍。

1. 查看一个空 Redis 实例的内存消耗：

```
127.0.0.1:6379> INFO MEMORY
```

```
# Memory
```

```
used_memory:827512
```

```
used_memory_human:808.12K
```

2. 我们最容易想到的方法是，把每种信息存储为字符串类型的键值对：

```
$ cat populatedata0.sh
#!/bin/bash
DATAFILE="string.data"
rm $DATAFILE >/dev/null 2>&1
NAMEOPTION[0]="Jack"
NAMEOPTION[1]="MIKE"
NAMEOPTION[2]="Mary"
SEXOPTION[0]="m"
SEXOPTION[1]="f"
NATIONOPTION[0]="us"
NATIONOPTION[1]="cn"
NATIONOPTION[2]="uk"
for i in `seq -f "%010g" 1 10000`
do
    namerand=$(( $RANDOM % 3 ))
    sexrand=$(( $RANDOM % 2 ))
    timerand=$(( $RANDOM % 30 ))
    nationrand=$(( $RANDOM % 3 ))
    echo "set \"user:${i}:name\" \"${NAMEOPTION[$namerand]}\" >> $DATAFILE
    echo "set \"user:${i}:sex\" \"${SEXOPTION[$sexrand]}\" >> $DATAFILE
    echo "set \"user:${i}:resigter_time\" \"date +%s%N\" >> $DATAFILE
    echo "set \"user:${i}:nation\" \"${NATIONOPTION[$nationrand]}\" >>
        $DATAFILE
    sleep 0.00000${timerand}
done
unix2dos $DATAFILE
$ bin/redis-cli FLUSHALL
OK
$ bash populatedata0.sh
unix2dos: converting file string.data to DOS format ...
$ cat string.data | redis-cli --pipe
```



```
All data transferred. Waiting for the last reply...
```

```
Last reply received from server.
```

```
errors: 0, replies: 40000
```

3. 通过 redis-cli, 使用 INFO MEMORY 命令查看内存消耗:

```
127.0.0.1:6379> INFO MEMORY
```

```
# Memory
```

```
used_memory:4231960
```

```
used_memory_human:4.04M
```

4. 我们要做的第一个优化是将一个用户的所有信息存储为一个 JSON 字符串:

```
$ cat populatedata1.sh
```

```
#!/bin/bash
```

```
DATAFILE="json.data"
```

```
... (omit all the variables declared above)
```

```
for i in `seq -f "%010g" 1 10000`
```

```
do
```

```
    namerand=$(( $RANDOM % 3 )
```

```
    sexrand=$(( $RANDOM % 2 )
```

```
    timerand=$(( $RANDOM % 30 )
```

```
    nationrand=$(( $RANDOM % 3 )
```

```
    echo "set \"user:${i}\" '{\"name\": \"${NAMEOPTION[$namerand]}\", sex: \"${
```

```
        SEXOPTION[$sexrand]}\", resigter_time: `date +%s%N`, nation: \"${
```

```
        NATIONOPTION[$nationrand]}\"}'" >> $DATAFILE
```

```
    sleep 0.00000${timerand}
```

```
done
```

```
unix2dos $DATAFILE
```

```
$ bin/redis-cli FLUSHALL
```

```
OK
```

```
$ bash populatedata1.sh
```

```
unix2dos: converting file string.data to DOS format ...
```

```
$ cat json.data | redis-cli --pipe
```

```
All data transferred. Waiting for the last reply...
```

```
Last reply received from server.
```

```
errors: 0, replies: 10000
```

5. 通过查看内存消耗，我们发现节省了约 43% $((4.04-2.29)/4.04)$ 的内存空间：

```
127.0.0.1:6379> INFO MEMORY
```

```
# Memory
```

```
used_memory:2398896
```

```
used_memory_human:2.29M
```

6. 在此之后，我们使用第 3 章数据特性中介绍的 Lua 脚本，利用 msgpack 库序列化原始的 JSON 字符串：

```
$ cat setjsonasmsgpack.lua
```

```
--EVAL 'this script' 1 some-key '{"some": "json"}'
```

```
local key = KEYS[1];
```

```
local value = ARGV[1];
```

```
local mvalue = cmsgpack.pack(cjson.decode(value));
```

```
return redis.call('SET', key, mvalue);
```

```
$ cat populatedata2.sh
```

```
#!/bin/bash
```

```
DATAFILE="msgpack.data"
```

```
rm $DATAFILE >/dev/null 2>&1
```

```
... (省略所有变量的声明)
```

```
for i in `seq -f "%010g" 1 10000`
```

```
do
```

```
    namerand=${RANDOM} % 3 ]
```

```
    sexrand=${RANDOM} % 2 ]
```

```
    timerand=${RANDOM} % 30 ]
```

```
    nationrand=${RANDOM} % 3 ]
```

```
    echo "user:00000${i}{\name\":"${NAMEOPTION[$namerand]}\",\sex\":"${SEXOPTION[$sexrand]}\",\resigter_time\":"date +%s`,\nation\":"${NATIONOPTION[$nationrand]}\\" }" >> $DATAFILE
```

```
    sleep 0.00000${timerand}
```

```
done
```

```
unix2dos $DATAFILE
```

```
$ bin/redis-cli FLUSHALL
```

```
OK
```

```
$ bash populatedata2.sh
```

```
unix2dos: converting file msgpack.data to DOS format ...
```

```
$ cat msgpack.data | while read CMD; do var1=$(cut -d' ' -f1 <<< $CMD); var2=
$(cut -d' ' -f2 <<< $CMD) ; /bin/redis-cli --eval setjsonasmmsgpack.lua
$var1 , $var2; done
```

7. 我们使用 msgpack 进行序列化的方案节省了约 49% $((4.04-2.06)/4.04)$ 的内存空间：

```
127.0.0.1:6379> INFO MEMORY
```

```
# Memory
```

```
used_memory:2159048
```

```
used_memory_human:2.06M
```

8. 然后，我们继续尝试使用哈希类型对设计进行优化：

```
$ cat populatedata3.sh
```

```
#!/bin/bash
```

```
DATAFILE="hash.data"
```

```
... (omit all the variables declared above)
```

```
for i in `seq -f "%010g" 1 10000`
```

```
do
```

```
namerand=$(( $RANDOM % 3 )
```

```
sexrand=$(( $RANDOM % 2 )
```

```
timerand=$(( $RANDOM % 30 )
```

```
nationrand=$(( $RANDOM % 3 )
```

```
echo "hset \"user:${i}\" \"name\" \"${NAMEOPTION[$namerand]}\" \"sex\" \"${
SEXOPTION[$sexrand]}\" \"resigter_time\" `date +%s%N` \"nation\" \"${
NATIONOPTION[$nationrand]}\" >> $DATAFILE
```

```
sleep 0.00000${timerand}
```

```
done
```



```
unix2dos $DATAFILE
```

```
$ bin/redis-cli FLUSHALL
```

```
OK
```

```
$ bash populatedata3.sh
```

```
unix2dos: converting file hash.data to DOS format ...
```

```
$ cat hash.data | redis-cli --pipe
```

```
All data transferred. Waiting for the last reply...
```

```
Last reply received from server.
```

```
errors: 0, replies: 10000</b>
```

9. 再次查看内存使用情况。然而很不幸，与之前使用 msgpack 的优化机制相比，该方案反而消耗了更多的内存，几乎与使用 JSON 字符串的设计方案所消耗的内存空间相同：

```
127.0.0.1:6379> INFO MEMORY
```

```
# Memory
```

```
used_memory:2399352
```

```
used_memory_human:2.29M
```

10. 我们发现，虽然使用了哈希来存储用户数据，但是与使用 JSON 字符串的方案相比，键的数量并没有被减少。因此，我们继续尝试通过对用户 ID 进行分区的方式来减少键的数量。首先，让我们修改哈希数据类型底层 ziplist 的配置参数 hash-max-ziplist-entries 和 hash-max-ziplist-value：

```
$ vim conf/redis.conf
```

```
hash-max-ziplist-entries 1000
```

```
hash-max-ziplist-value 64
```

11. 重新启动 Redis 服务器使配置生效。然后，验证我们的设计原型：

```
$ cat populatedata4.sh
```

```
#!/bin/bash
```

```
DATAFILE="hashpartition.data"
```

```
PLENGTH=3
```

```
...（省略所有变量的声明）
```

```

for i in `seq -f "%010g" 1 10000`
do
    namerand=${ $RANDOM % 3 }
    sexrand=${ $RANDOM % 2 }
    timerand=${ $RANDOM % 30 }
    nationrand=${ $RANDOM % 3 }
    LENGTH=`echo ${#i}`
    LENGTHCUT=`echo ${ (LENGTH-PLENGTH) }`
    LENGTHEND=`echo ${ (LENGTHCUT+1) }`
    VALUE1=`echo $i | cut -c1-${LENGTHCUT}`
    VALUE2=`echo $i | cut -c${LENGTHEND}-${LENGTH}`
    echo "hset \"user:${VALUE1}\" $VALUE2 '{\"name\": \"${NAMEOPTION[$namerand]}\", \"sex\": \"${SEXOPTION[$sexrand]}\", \"resigter_time\": `date +%s%N`, \"nation\": \"${NATIONOPTION[$nationrand]}\", \"}\"'" >> $DATAFILE
    sleep 0.00000${timerand}
done

unix2dos $DATAFILE

$ bin/redis-cli FLUSHALL
OK

$ bash populatedata4.sh
unix2dos: converting file hashpartition.data to DOS format ...

$ cat hashpartition.data | redis-cli --pipe
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 10000</b>

```

12. 通过查看哈希分区方案的内存消耗情况，我们可以清楚地看到，这种优化方案节省了约 52% $((4.04-1.94)/4.04)$ 的内存空间：

```

127.0.0.1:6379> INFO MEMORY
# Memory
used_memory:2032160
used_memory_human:1.94M

```

13. 最后，我们尝试将哈希分区和 msgpack 序列化方案结合起来，看看效果如何：

```
$ cat populatedata5.sh
#!/bin/bash
DATAFILE="hashpartitionmsgpack.data"
PLENGTH=3

rm $DATAFILE >/dev/null 2>&1

... (省略所有变量的声明)
for i in `seq -f "%010g" 1 10000`
do
    namerand=$(( $RANDOM % 3 ])
    sexrand=$(( $RANDOM % 2 ])
    timerand=$(( $RANDOM % 30 ])
    nationrand=$(( $RANDOM % 3 ])
    LENGTH=`echo ${#i}`
    LENGTHCUT=$((LENGTH-PLENGTH))
    LENGTHEND=$((LENGTHCUT+1))
    VALUE1=`echo $i | awk '{print substr($1,1,"'$LENGTHCUT'")}'`
    VALUE2=`echo $i | awk '{print substr($1,"'$LENGTHEND'","'$PLENGTH'")}'`
    echo "user:${VALUE1} $VALUE2 {"name\:\"${NAMEOPTION[$namerand]}\", \"sex
    \":\"${SEXOPTION[$sexrand]}\", \"resigter_time\:\"date +%s%N\", \"nation\:\"
    ${NATIONOPTION[$nationrand]}\"}" >> $DATAFILE sleep 0.00000${timerand}
done

unix2dos $DATAFILE

$ bin/redis-cli FLUSHALL
OK

$ bash populatedata5.sh
unix2dos: converting file hashpartitionmsgpack.data to DOS format ...

$ cat hashpartitionmsgpack.data | while read CMD; do var1=$(cut -d' ' -f1 <<<
    $CMD); var2=$(cut -d' ' -f2 <<< $CMD); var3=$(cut -d' ' -f3 <<< $CMD); /
    redis/bin/redis-cli --eval setjsonashashmsgpack.lua $var1 $var2 , $var3;
done
```


14. 结果让人大吃一惊：在采取了这种优化后，样本数据只需要消耗 1.34M 内存空间，也就仅仅是原始设计的 33%(1.34/4.04)：

```
127.0.0.1:6379> INFO MEMORY
# Memory
used_memory:1403272
used_memory_human:1.34M
```

4.3.3 工作原理

在我们设计一个使用 Redis 的应用程序时，应该记住的一点是，通常并不只有一种数据类型适用于我们的业务场景。这意味着，对存储和查询性能的不同要求会促成不同的设计。在设计时，我们的工作是通过像前面案例一样在各种各样的原型验证中确定哪种数据类型最为合适。

在本例中，我们首先做了一个基本的设计，也就是简单地把一个用户的每个属性值映射到一个键中。该基本设计的内存消耗情况被作为后续设计的参考基线。

在此之后，我们尝试了通过两种不同的方法来降低内存的消耗。一种方法是尝试减少键的数量。我们之所以这样做的原因在于，Redis 使用了一些内部数据结构来维护一个键。通过减少键的数量，内部数据结构占用的空间更少，从而可以节省大量的内存空间。出于这种考虑，我们在此前的案例中依次尝试了使用 JSON 字符串作为一个用户的值和使用哈希类型两种方法。

哈希键的分区应该引起特别注意。因为我们想在一个哈希键中存储 1000 个元素，所以要做的第一件事就是将配置项 `hash-max-ziplist-entries` 的值修改为 1000，以确保哈希键的值使用 `ziplist` 的内部编码方式。我们在第 2 章数据类型一章使用哈希 (*hash*) 类型的案例中已经学习过，如果 Redis 中哈希的长度小于 `hash-max-ziplist-entries` 且列表中的每个元素的大小都小于 `hash-max-ziplist-value`，那么会使用存储效率更高的 `ziplist` 作为哈希对象的编码方式。在配置文件修改后，需要重新启动 Redis 服务器以使配置生效。我们按照用户 ID 的前七个数字对键进行分区，而其余的用户信息则以 JSON 字符串的形式存储。因此，每个分区包含 1000 个元素（"user:0000000" 和 "user:0000010" 除外），其组织方式如下：

```
$ bin/redis-cli hgetall "user:0000007"
088
{"name":"Mary","sex":"f","resigter_time":1506942816344887079,"nation":"uk"}
...
331
{"name":"Mary","sex":"f","resigter_time":1506942817274215585,"nation":"us"}

$ bin/redis-cli hlen "user:0000007"
```

```
(integer) 1000
```

这个案例最后只需要 11 个键，比原始方案所需的键数（40000）要少得多：

```
$ bin/redis-cli dbsize
```

```
(integer) 11
```

另一种减少内存消耗的方法是压缩键的值。Lua 脚本中的 `msgpack` 库在这方面发挥了重要的作用。

我们最终的设计则结合了这两种设计。

另一种减少内存消耗的简单方法是缩小键的长度。例如，我们可以把 `regt` 作为 "register_time" 的缩写。读者可以试试这个策略，看看能够节省多少内存空间。

4.3.4 更多细节

我们还可以举出更多的例子，比如计算网站的独立访客数。对于这个需求，我们可以使用集合、位图或 HyperLogLog 来实现。对于此类的应用设计，我们必须选择正确的数据类型；对 Redis 的数据类型越熟悉，就越能找到更多的方法来实现应用程序。

除了内存消耗问题之外，我们还应该注意操作所选择的数据类型时的性能问题。我们确实必须不断地考虑时间-空间的平衡；但也请注意，不要对内存消耗问题进行过度的设计。有关使用 Redis 的应用程序设计中需要考虑的性能问题，在本章的相关内容小节中提供了一些有用的建议。

4.3.5 相关内容

- 有关更多内存优化的细节，请参阅：<https://redis.io/topics/memory-optimization>。
- Instagram 的技术团队博客描述了一个成功节省内存的用例，请参阅：<https://engineering.instagram.com/storing-hundreds-of-millions-of-simple-key-value-pairs-in-redis-1091ae80f74c>。
- Deliveroo（一家英国的在线食品配送公司）的技术团队博客描述了另一个在 Redis 中存储会话的成功实践，请参阅：<https://deliveroo.engineering/2016/10/07/optimising-session-key-storage.html>。

4.4 使用正确的 API

在第 2 章数据类型和第 3 章数据特性中，我们看到 Redis 具有丰富的数据类型，并提供了许多强大的 API 来操作它们。当我们使用 Redis 开发应用程序时，与此前使用正确的数据类型的案例中所提到的数据类型的选择类似，总会发现不只一个 API 能够实现某些业务需求。为了保证 Redis 实例的性能，在决定使用哪个 API 时必须非常谨慎。

在本案例中，我们将通过几个例子来学习如何在 Redis 中使用正确的 API 获得良好的性能。

4.4.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

在学习下一节的内容之前，我们需要使用 `FLUSHALL` 命令清除 Redis 实例中的所有数据。

在 Ubuntu 中，我们可以使用以下的命令安装 `dos2unix` 工具：

```
sudo apt-get install dos2unix
```

在 macOS 中，我们可以使用以下的命令安装 `dos2unix` 工具：

```
brew install dos2unix
```

4.4.2 操作步骤

接下来，让我们按照如下的步骤来学习如何在 Redis 中使用正确的 API：

1. 在本案例的开头，我们通过两个 API（`HSET` 和 `HMSET`）把 100 万个用户的数据导入到了一个哈希类型的键中。为了更好地进行演示，我们会执行以下的脚本，这个脚本会连接到 Redis 服务器，然后向另一台机器（应用服务器）中导入用户数据。我们可以运行这个脚本然后休息一会儿，回头再看结果：

```
$ cat hmset-vs-hset.sh
#!/bin/bash
HSETFILE="hset.cmd"
HMSETFILE="hmset.cmd"

rm $HSETFILE $HMSETFILE

... (省略所有变量的声明)

printf "hmset \"user\" \" \" >> $HMSETFILE
for i in `seq -f "%010g" 1 1000000`
do
    namerand=$(( $RANDOM % 3 ])
    sexrand=$(( $RANDOM % 2 ])
    timerand=$(( $RANDOM % 30 ])
    nationrand=$(( $RANDOM % 3 )]
```



```

echo "hset \"user\" ${i} '{\"name\": \"${NAMEOPTION[$namerand]}\", \"sex
\": \"${SEXOPTION[$sexrand]}\", \"resigter_time\": `date +%s%N`, \"nation
\": \"${NATIONOPTION[$nationrand]}\"}'" >> $HSETFILE
printf " ${i} '{\"name\": \"${NAMEOPTION[$namerand]}\", \"sex\": \"${SEXOPTION[
$sexrand]}\", \"resigter_time\": `date +%s%N`, \"nation\": \"${NATIONOPTION[
$nationrand]}\"}'" >> $HMSETFILE
sleep 0.00000${timerand}
done

unix2dos $HSETFILE
unix2dos $HMSETFILE

time cat $HSETFILE | /redis/bin/redis-cli -h $SERVER
sleep 10
time cat $HMSETFILE | /redis/bin/redis-cli -h $SERVER

$ bin/redis-cli FLUSHALL
OK

$ bash hmsset-vs-hset.sh
...

real    0m4.557s
user    0m0.696s
sys     0m1.284s
OK
(0.51s)

real    0m0.750s
user    0m0.224s
sys     0m0.496s

```

2. 数据导入完成后，我们尝试使用 `HGETALL` 命令来获取哈希键“user”的所有数据（该命令会花费一些时间，因为对于 Redis 而言，一个秒级的操作已经可以被算作是运行很久了）。在使用 `HGETALL` 命令之前，我们打开另一个终端，然后使用 `redis-cli` 的 `--latency` 选项来进行延迟测试：

```
$ bin/redis-cli --latency
min: 0, max: 1, avg: 0.11 (136 samples)
```

3. 通过 HGETALL 命令获取所有用户数据：

```
$ bin/redis-cli HGETALL user
```

4. 在 HGETALL 命令的处理过程中会检测到很高的延迟：

```
$ bin/redis-cli --latency
min: 0, max: 57, avg: 0.13 (2474 samples)
```

5. 按 *Ctrl+C* 停止延迟测试，然后重新启动：

```
$ bin/redis-cli --latency
min: 0, max: 1, avg: 0.08 (186 samples)
```

6. 我们尝试使用 HSCAN 命令而不是使用 HGETALL 命令来遍历所有用户数据：

```
$ cat hscan.sh
#!/bin/bash
cr=0
key=$1

rm ${1}.dumpfile

while true; do
    cr=~/.redis/bin/redis-cli HSCAN user $cr MATCH '*' | {
        read a
        echo $a
        while read x; read y; do
            echo $x:$y >> ${1}.dumpfile
        done
    }`

    echo $cr

    if [ $cr == "0" ]; then
        break
    fi
fi
```

```
done
```

```
$ bash hscan.sh user
```

7. 在遍历过程中，没有出现延迟太高的情况：

```
$ bin/redis-cli --latency
```

```
min: 0, max: 1, avg: 0.09 (13423 samples)
```

4.4.3 工作原理

当我们准备设计一个使用 Redis 的应用程序时，应该考虑两个原则：第一个原则是，应该尽一切努力将数据操作组合在一起降低往返时延（RTT）。当我们在第 3 章数据特性中使用管道的案例中讨论管道的特性时引入了往返时延的概念。Redis 以高速处理请求而闻名。因此，如果我们能够降低往返时延，那么将获得显著的性能提升。利用管道的优点确实是一个好主意，而一些 Redis 的数据 API 还可以原生地降低往返时延。实际上，上例中第一部分正是对这一原则的良好演示。与使用 HSET 命令逐个地设置哈希值相比，如果所有的数据都可以在设置之前准备好，那么使用 HMSET 命令在一次网络通信中设置所有的哈希值显然是一个更好的方法。事实上，这种方法能够节省 83.5% ($(1-0.750/4.557)$) 的时间！

我们应该牢记的另一个原则是，Redis 本质上是一个单线程的数据存储服务，这意味着我们在选择 Redis 命令时应该非常谨慎，并牢记这些命令的时间复杂度。Redis 的文档提供了每个 API 的时间复杂度。例如，我们可以在 Redis 文档中找到 HGETALL 命令的时间复杂度，如下：

时间复杂度： $O(N)$

其中， N 是哈希的大小。

这里，我们使用大 O 符号来表示时间复杂度。我们只需要明白该符号是一个被广泛用于描述算法性能或复杂性的术语即可，无需对其他细节进行过多的讨论。为了便于读者直观地认识，图 4.1 展示了常见大 O 符号的复杂度的增长速度。

在我们的示例中，在将数据放到 Redis 的一个哈希类型的键中后，我们尝试了使用 HGETALL 命令来获取所有的数据。这样做引发了可怕的延迟，而严重的延迟通常正是一个在线 Redis 数据服务的噩梦。发生这种情况的原因是 HGETALL 命令的时间复杂度为 $O(n)$ ，其中 n 是哈希的大小。在我们的示例中， n 是 100000，这个数字相对于 Redis 的处理速度来说已经是一个非常大的数字了。在处理此命令期间，Redis 服务器无法响应任何其他请求。实现相同目标的更好方法是，使用 HSCAN 命令遍历哈希类型的键。这个命令会渐进地遍历哈希类型的键，从而有效地减少了延迟的激增。

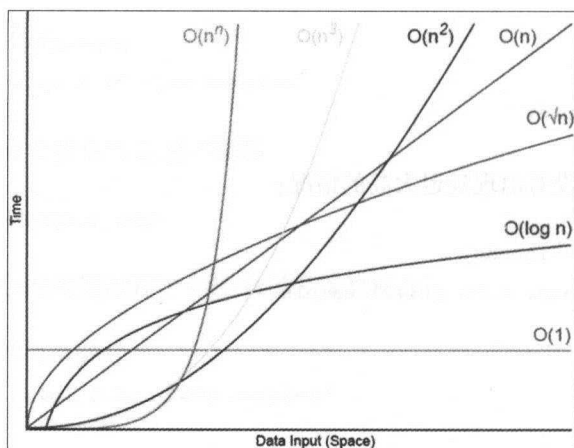


图 4.1 复杂度的增长速度

(来源: <https://github.com/sf-wdi-31/algorithm-complexity-and-big-o>)

4.4.4 更多细节

此外, 诸如 `KEYS *`、`FLUSHDB`、`HDEL` 和 `HDEL` 等的命令都可能会阻塞 Redis 服务器。一般来说, 我们在使用那些时间复杂度高于 $O(n)$ 的 API 时应该特别谨慎。

4.4.5 相关内容

- 如果想知道究竟是什么操作拖慢了 Redis 服务器的响应速度, 可以使用慢日志来记录 Redis 服务器中处理的慢命令。有关详细信息, 请参阅第 10 章 *Redis 故障诊断* 中使用慢日志定位慢操作/查询的案例。
- 有关更多延迟的故障诊断机制将在第 10 章 *Redis 故障诊断* 中延迟问题的故障诊断的案例中介绍。
- 对于大 O 符号的更多细节, 请参阅: https://en.wikipedia.org/wiki/Big_O_notation。

4.5 使用 Java 连接到 Redis

我们需要使用 Redis 的 Java 客户端才能在 Java 应用程序中使用 Redis。在 Redis 的官网的 `Clients` 栏目中, 列举了一些可以选择的 Java 客户端。在本案例中, 我们将介绍 Jedis 这个开源且易用的 Redis Java 客户端。

4.5.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。

我们需要安装 1.8 版本的 **Java 开发工具包（Java Development Kit, JDK）**。

建议读者使用诸如 IntelliJ IDEA 或 NetBeans 的 Java IDE，但并不是必需的。

4.5.2 操作步骤

接下来，让我们按照一下的步骤来学习如何使用 Java 连接到 Redis。

首先，我们需要在 Java 应用程序的项目中引用 Jedis 对应的库。我们既可以下载 Jedis 库的 JAR 包并将其添加到 CLASSPATH 中，也可以通过诸如 Maven、Gradle 或 Bazel 之类的构建工具来管理依赖。在本案例中，我们使用 Gradle 来管理 Jedis 的依赖项。

将下面一行添加到 `build.gradle` 的 `dependencies` 部分：

```
compile group: 'redis.clients', name: 'jedis', version: '2.9.0'
```

连接到 Redis 服务器

使用下列代码创建一个 Java 类，`JedisSingleDemo`：

```
1  import redis.clients.jedis.Jedis;
2  import java.util.List;
3
4  public class JedisSingleDemo {
5      public static void main(String[] args) {
6          //Connecting to localhost Redis server
7          Jedis jedis = new Jedis("localhost");
8
9          //String operations
10         String restaurant = "Extreme Pizza";
11         jedis.set(restaurant, "300 Broadway, New York, NY");
12         jedis.append(restaurant, " 10011");
13         String address = jedis.get("Extreme Pizza");
14         System.out.printf("Address for %s is %s\n", restaurant, address);
15
16         //List operations
17         String listKey = "favorite_restaurants";
18         jedis.lpush(listKey, "PF Chang's", "Olive Garden");
19         jedis.rpush(listKey, "Outback Steakhouse", "Red Lobster");
20         List<String> favoriteRestaurants = jedis.lrange(listKey, 0, -1);
21         System.out.printf("Favorite Restaurants: %s\n", favoriteRestaurants);
22
23         System.exit(0);
24     }
25 }
```

示例程序将输出如下的内容：

```
Address for Extreme Pizza is 300 Broadway, New York, NY 10011
Favorite Restaurants: [Olive Garden, PF Chang's, Olive Garden, PF Chang's,
    Indian Tandoor, Longhorn Steakhouse, Outback Steakhouse, Red Lobster,
    Outback Steakhouse, Red Lobster]
```

在 Jedis 中使用管道

创建另一个 Java 类，JedisPipelineDemo：

```
1 import redis.clients.jedis.Jedis;
2 import redis.clients.jedis.Pipeline;
3 import redis.clients.jedis.Response;
4
5 public class JedisPipelineDemo {
6     public static void main(String[] args) {
7         //Connecting to localhost Redis server
8         Jedis jedis = new Jedis("localhost");
9
10        //Create a Pipeline
11        Pipeline pipeline = jedis.pipelined();
12        //Add commands to pipeline
13        pipeline.set("mykey", "myvalue");
14        pipeline.sadd("myset", "value1", "value2");
15        Response<String> stringValue = pipeline.get("mykey");
16        Response<Long> noElementsInSet = pipeline.scard("myset");
17        //Send commands
18        pipeline.sync();
19        //Handle responses
20        System.out.printf("mykey: %s\n", stringValue.get());
21        System.out.printf("Number of Elements in set: %d\n", noElementsInSet.get());
22        System.exit(0);
23    }
24 }
```

该程序的输出为：

```
mykey: myvalue
Number of Elements in set: 2
```

在 Jedis 中使用事务

创建一个 Java 类，JedisTransactionDemo：


```

1 import redis.clients.jedis.Jedis;
2 import redis.clients.jedis.Response;
3 import redis.clients.jedis.Transaction;
4 import java.util.Set;
5
6 public class JedisTransactionDemo {
7     public static void main(String[] args) {
8         //Connecting to localhost Redis server
9         Jedis jedis = new Jedis("localhost");
10
11         //Initialize
12         String user = "user:1000";
13         String restaurantOrderCount = "restaurant_orders:200";
14         String restaurantUsers = "restaurant_users:200";
15         jedis.set(restaurantOrderCount, "400");
16         jedis.sadd(restaurantUsers, "user:302", "user:401");
17
18         //Create a Redis transaction
19         Transaction transaction = jedis.multi();
20         Response<Long> countResponse = transaction.incr(restaurantOrderCount);
21         transaction.sadd(restaurantUsers, user);
22         Response<Set<String>> userSet = transaction.smembers(restaurantUsers);
23         //Execute transaction
24         transaction.exec();
25
26         //Handle responses
27         System.out.printf("Number of orders: %d\n", countResponse.get());
28         System.out.printf("Users: %s\n", userSet.get());
29         System.exit(0);
30     }
31 }

```

该程序的输出为：

```
Number of orders: 401
```

```
Users: [user:1000, user:401, user:302]
```

在 Jedis 中运行 Lua 脚本

创建一个名为 `updateJson.lua` 的 Lua 脚本，并将其放入 `Java resources` 文件夹中。Lua 脚本的内容可以在第 3 章数据特性中使用 `Lua` 脚本的案例中找到。

创建一个 Java 类，`JedisLuaDemo`：该程序的输出为：

```
users:id:992452: {"grade":"C","name":"Tina","sex":"female"}
```

```

1 package com.packtpub.redis_cookbook;
2
3 import redis.clients.jedis.Jedis;
4 import java.io.BufferedReader;
5 import java.io.InputStream;
6 import java.io.InputStreamReader;
7 import java.util.Collections;
8 import java.util.List;
9 import java.util.stream.Collectors;
10
11 public class JedisLuaDemo {
12     public static void main(String[] args) throws Exception {
13         //Connecting to localhost Redis server
14         Jedis jedis = new Jedis("localhost");
15
16         String user = "users:id:992452";
17         jedis.set(user, "{\"name\": \"Tina\", \"sex\": \"female\", \"grade\": \"A\"}");
18
19         //Register Lua script
20         InputStream luaInputStream =
21             JedisLuaDemo.class
22                 .getClassLoader()
23                 .getResourceAsStream("updateJson.lua");
24         String luaScript =
25             new BufferedReader(new InputStreamReader(luaInputStream))
26                 .lines()
27                 .collect(Collectors.joining("\n"));
28         String luaSHA = jedis.scriptLoad(luaScript);
29
30         //Eval Lua script
31         List<String> KEYS = Collections.singletonList(user);
32         List<String> ARGS = Collections.singletonList("{\"grade\": \"C\"}");
33         jedis.evalsha(luaSHA, KEYS, ARGS);
34
35         System.out.printf("%s: %s\n", user, jedis.get(user));
36         System.exit(0);
37     }
38 }

```

在 Jedis 中使用连接池

创建一个 Java 类，JedisPoolDemo。

该程序的输出为：

```

Kyoto Ramen rating: 5.0
rating: 5.0
phone: 555-123-6543
address: 801 Mission St, San Jose, CA

```

```

1  import redis.clients.jedis.Jedis;
2  import redis.clients.jedis.JedisPool;
3  import redis.clients.jedis.JedisPoolConfig;
4
5  import java.util.HashMap;
6  import java.util.Map;
7
8  public class JedisPoolDemo {
9      public static void main(String[] args) {
10         //Creating a JedisPool of Jedis connections to localhost Redis server
11         JedisPool jedisPool = new JedisPool(new JedisPoolConfig(), "localhost");
12
13         //Get a Jedis connection from pool
14         try (Jedis jedis = jedisPool.getResource()) {
15             String restaurantName = "Kyoto Ramen";
16             Map<String, String> restaurantInfo = new HashMap<>();
17             restaurantInfo.put("address", "801 Mission St, San Jose, CA");
18             restaurantInfo.put("phone", "555-123-6543");
19             jedis.hmset(restaurantName, restaurantInfo);
20             jedis.hset(restaurantName, "rating", "5.0");
21             String rating = jedis.hget(restaurantName, "rating");
22             System.out.printf("%s rating: %s\n", restaurantName, rating);
23             //Print out hash
24             for (Map.Entry<String, String> entry: jedis.hgetAll(restaurantName).entrySet()) {
25                 System.out.printf("%s: %s\n", entry.getKey(), entry.getValue());
26             }
27         }
28         System.exit(0);
29     }
30 }

```

4.5.3 工作原理

在第一个示例 `JedisSingleDemo` 中，我们通过创建带有服务器主机名的 `Jedis` 实例连接到了 Redis 服务器。如果未指定端口的话，默认连接的服务器端口为 6379。`Jedis` 类还有其他的几个构造函数，通过这些构造函数可以指定服务器端口、连接超时时间等。

`Jedis` 实例创建好后，我们就可以调用该实例的方法向 Redis 服务器发送命令了。方法的名字与前面几章中介绍的 Redis 命令的名字相同。

在第 3 章数据特性中使用管道的案例中，我们学习了如何通过手工构造原始的 RESP 字符串来使用 Redis 的管道功能。在 `Jedis` 的帮助下，使用 Redis 的管道功能要容易得多，因为 `Jedis` 会负责为我们构建原始的 RESP 字符串。

在 `JedisPipelineDemo` 示例中，我们使用 `pipelined()` 方法创建了一个 `Jedis` 管道实例。之后，我们就可以像直接执行命令那样向管道中添加命令了。这里的区别在于，所添加的命令的响应是 `Response<T>` 类型的对象。在将管道中的命令发送到服务器并执行前，这些对象是不可用的。我们可以用 `sync()` 方法将管道中的命令发送到服务器；之后，我们就可以获取响应的内容了。

与管道的创建类似，`JedisTransactionDemo` 示例中使用 `multi()` 方法创建了一个 Redis 事务实例，并使用 `exec()` 方法执行事务。在事务执行结束后，我们才可以使用响应返回的结果。

在 `JedisLuaDemo` 示例中,我们首先从 `resources` 文件夹中以字符串的形式读取文件 `updateJsc.lua`, 然后使用 `scriptLoad()` 注册 Lua 脚本并得到 SHA。随后, 我们使用 `evalsha()` 执行 Lua 脚本; `KEYS` 和 `ARGS` 都是 `List<String>` 类型的, 并被作为参数传给 `evalsha()`。

我们在前述示例中创建的 `Jedis` 实例不是线程安全的, 这意味着不同的线程不应该共享相同的 `Jedis` 实例。除了为每个线程创建 `Jedis` 实例外, 我们还可以使用线程安全的连接池, `JedisPool`。当我们需要连接的时候, 直接从连接池中获取一个连接; 当我们使用完连接后, 将其返还回到连接池即可。这样, 我们就可以节省创建和关闭连接的开销 (因为池中的连接都是已经建立好的)。

从 `JedisPool` 中获取的连接在使用完后必须通过调用 `close()` 返还回到连接池中。因为 `Jedis` 类实现了 `AutoClosable` 接口, 所以我们可以使用 `try-resource` 块来关闭连接并将其返还回到连接池中。

4.5.4 相关内容

- 本案例只是简单地就在 Java 应用程序中使用 `Jedis` 进行了说明。`Jedis` 是一个开源项目, 对于我们未在本案例中涉及的内容 (`pub/sub`、复制等等), 请参阅 `Jedis` 的 GitHub 页面:
<https://github.com/xetorthio/jedis>。

4.6 使用 Python 连接到 Redis

对于使用 Python 连接到 Redis 来说, 我们也有几种客户端可以选择。在本案例中, 我们将简要地介绍如何使用 Python 的 Redis 客户端, `redis-py`。

4.6.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器, 并使用 `redis-cli` 连接到这个 Redis 服务器。

我们需要安装 Python 2.6+ 或 Python 3.4+。

4.6.2 操作步骤

下面, 让我们学习如何使用 Python 连接到 Redis。首先, 我们需要安装 `redis-py` 库。我们可以通过 PyPI 很容易地安装 `redis-py`, 只需要运行以下命令即可:

```
pip install redis
```

连接到 Redis 服务器

使用下列代码创建文件 RedisDemo.py:

```
1  from __future__ import print_function
2  import redis
3
4  # Create connection to localhost Redis
5  client = redis.StrictRedis(host="localhost", port=6379)
6
7  # String Operations
8  restaurant = "Extreme Pizza"
9  client.set(restaurant, "300 Broadway, New York, NY")
10 client.append(restaurant, " 10011")
11 address = client.get("Extreme Pizza")
12 print("Address for " + restaurant + " is: " + address)
13
14 # List operations
15 listKey = "favorite_restaurants"
16 client.lpush(listKey, "PF Chang's", "Olive Garden")
17 client.rpush(listKey, "Outback Steakhouse", "Red Lobster")
18 favoriteRestaurants = client.lrange(listKey, 0, -1)
19 print("Favorite Restaurants: ", favoriteRestaurants)
```

示例程序将输出如下的内容:

```
Address for Extreme Pizza is 300 Broadway, New York, NY 10011
```

```
Favorite Restaurants: [Olive Garden, PF Chang's, Olive Garden, PF Chang's,
    Indian Tandoor, Longhorn Steakhouse, Outback Steakhouse, Red Lobster,
    Outback Steakhouse, Red Lobster]
```

使用管道

Redis 管道可以很容易地通过 redis-py 实现。

创建另一个文件, RedisPipelineDemo.py:

```
1 from __future__ import print_function
2 import redis
3
4 # Create connection to localhost Redis
5 client = redis.StrictRedis(host="localhost", port=6379)
6
7 # Create a pipeline
8 pipeline = client.pipeline()
9
10 # Add commands to pipeline
11 pipeline.set("mykey", "myvalue")
12 pipeline.sadd("myset", "value1", "value2")
13 pipeline.get("mykey")
14 pipeline.scard("myset")
15
16 # Send commands
17 response = pipeline.execute()
18 print(response)
```

在本例中，我们将得到一个包括 SET、SADD、GET 和 SCARD 命令对应响应的列表：

```
[True, 0, 'myvalue', 2]
```

运行 Lua 脚本

redis-py 提供了一个非常方便的 `register_script()` 函数用于注册 Lua 脚本。`register_script()` 将返回一个 `Script` 实例，该实例用于后续调用 Lua 脚本。在第 3 章数据特性中使用管道的案例中，我们使用了 `SCRIPT LOAD` 和 `EVALSHA` 命令来缓存和复用 Lua 脚本。让我们看看这在 redis-py 中如何实现：

`updateJson.lua` 文件的内容可以在第 3 章数据特性中使用管道的案例中找到。

使用如下的代码创建文件 `RedisLuaDemo.py`。该程序的输出将是被更新后的 JSON：

```
{"grade": "C", "name": "Tina", "sex": "female"}
```



```

1  from __future__ import print_function
2  import redis
3
4  # Create connection to localhost Redis
5  client = redis.StrictRedis(host="localhost", port=6379)
6
7  user = "users:id:992452"
8  client.set(user, '{"name": "Tina", "sex": "female", "grade": "A"}')
9
10 # Read the lua scripts from file
11 with open("updateJson.lua") as f:
12     lua = f.read()
13
14 #Create Redis Script instance
15 updateJson = client.register_script(lua)
16
17 #Invoke lua script using the script instance
18 updateJson(keys=[user], args=['{"grade": "C"}'])
19
20 print(client.get(user))

```

4.6.3 工作原理

要使用 `redis-py` 库，我们首先需要使用 `"import redis"` 导入该模块。`RedisDemo.py` 文件第 6 行的 `"redis.StrictRedis()"` 创建了一个连接到本地主机 Redis 服务器的 `StrictRedis` 实例。`StrictRedis` 中的方法用于向 Redis 服务器发送命令。大多数方法的名称及其语法与我们在第 2 章数据类型和第 3 章数据特性中讲解的 Redis 命令相同。少数例外的情况稍后将在本案例中进行解释。

在 `RedisPipelineDemo.py` 中，`pipeline()` 方法会创建一个 Redis 管道实例。之后，我们可以按照与直接执行命令相同的方式向管道中添加命令。调用 `execute()` 会把管道中的命令发送给服务器，并按顺序返回命令的响应。

默认情况下，`redis-py` 会用 `MULTI` 和 `EXEC` 命令把管道中的命令封装成一个事务并以原子化的方式执行。我们可以通过将 `execute()` 方法的 `transaction` 参数设置为 `False` 来禁用这种行为：

```
pipeline.execute(transaction=False)
```

在 `RedisLuaDemo.py` 示例中，我们首先从一个外部文件中读取 Lua 脚本的内容，然后通过调用 `register_script()` 来注册脚本。`register_script()` 返回的实例可以当作一个函数直接调用；在 Lua 脚本中，我们可以使用 `KEYS[]` 和 `ARGV[]` 来将 `keys` 和 `args` 参数传递到该函数中来调用脚本。

4.6.4 更多细节

StrictRedis 类中的方法几乎都遵循 Redis 官方命令的命令名称和语法，但有几个例外：

1. 因为 del 是 Python 中的保留字，所以 DEL 命令被重命名为 delete。
2. StrictRedis 类中没有 MULTI/EXEC 命令，这两个命令是在 pipeline 类中实现的。
3. 由于线程安全问题，SELECT 命令没有实现。我们稍后将对此进行解释。
4. StrictRedis 类中没有 SUBSCRIBE/LISTEN 命令，这两个命令是在 PubSub 类中实现的。

在 redis-py 中，还有一个名为 Redis 的类。它是 StrictRedis 的一个子类，用于为老版本的 redis-py 提供向后兼容。它们之间的细微差别可以在 redis-py 的 GitHub 页面上找到。

redis-py 中的 StrictRedis 实例是线程安全的，因为其内部有一个管理到 Redis 服务器连接的连接池。在默认情况下，每个实例都有自己的连接池。在创建新的 StrictRedis 或 Redis 实例时，通过将连接池实例作为 connection_pool 参数传入可以覆盖这种默认行为。调用 redis.ConnectionPool() 可以创建连接池实例，但连接池实例必须在 StrictRedis 或 Redis 实例之前创建：

```
>>> connectionPool = redis.ConnectionPool(host="localhost", port=6379)
>>> client = redis.StrictRedis(connection_pool=connectionPool)
```

4.6.5 相关内容

- 本案例无法涵盖有关 redis-py 的所有细节。由于它是一个开源项目，请参阅其 GitHub 页面寻找有关这个库的更多细节：<https://github.com/andymccurdy/redis-py>。

4.7 使用 Spring Data 连接到 Redis

在前两个案例中，我们展示了如何在 Java 和 Python 中连接到 Redis。现在，让我们讨论如何在 Web 应用中使用 Redis。对于使用 Java 进行 Web 开发而言，Pivotal 的 Spring 是最为著名的框架，它利用 MVC 模式来快速地开发健壮的 Java Web 应用程序。通过 Spring Data Redis 库，我们可以在 Spring 中开箱即用地使用 Redis。

在本案例中，我们将创建一个实现了用户模型创建/读取/更新/删除（CRUD）操作的示例项目，以此来展示如何使用 Spring Data Redis 库来连接到 Redis。

4.7.1 准备工作

我们需要按照启动和停止 *Redis* 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。在学习这一节的内容之前，我们还需要使用 `FLUSHALL` 命令清除 Redis 实例中的所有数据。

本案例推荐使用一个 IDE。这里，我们使用的是 IntelliJ IDEA（社区版，简称为 IntelliJ）。读者可以从以下链接下载：<https://www.jetbrains.com/idea/download/>。

要编译和运行代码需要 JDK 8 或以上版本。

由于本书篇幅有限，我们不会讨论这个示例项目的所有细节，只有与 Redis 相关的代码和设置会在以下两部分中讨论。对于完整的项目，请参考本书配套的示例代码。

4.7.2 操作步骤

让我们按照以下的步骤学习使用 Spring Data 连接到 Redis：

1. 使用 Spring Initializer 在 IDEA 中创建一个新项目，如图4.2所示。

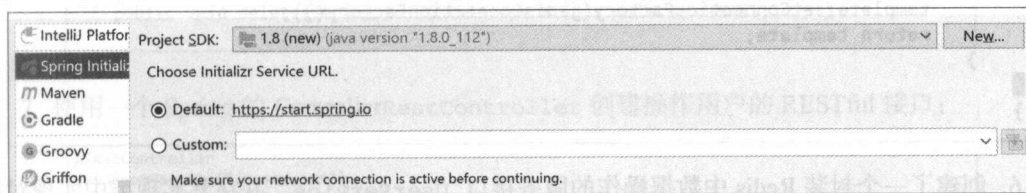


图 4.2 创建新项目

2. 点击 Next 继续，并填入项目的元数据。
3. 选择示例项目的依赖项，然后完成项目的创建，如图4.3所示：

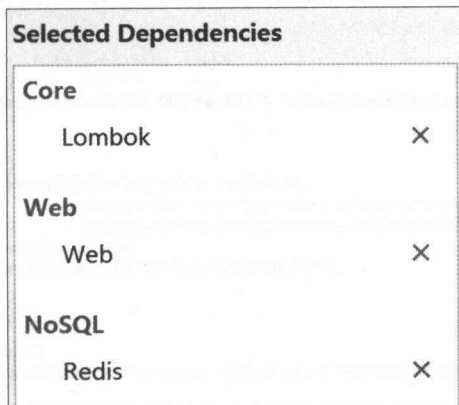


图 4.3 选择依赖项

4. 首先，为用户对象创建一个 Model 类 `User`，作为其 POJO 对象。

5. 之后，构造一个应用配置类，在其中加载 Redis 服务器的地址和端口，并创建 `RedisTemplate` 实例：

```
@Configuration
public class AppConfig {
    private @Value("${redis.host}") String redisHost;
    private @Value("${redis.port}") int redisPort;

    @Bean
    JedisConnectionFactory jedisConnectionFactory(){
        JedisConnectionFactory factory = new JedisConnectionFactory();
        factory.setHostName(redisHost);
        factory.setPort(redisPort);
        factory.setUsePool(true);
        return factory;
    }

    @Bean
    RedisTemplate<String, User> redisTemplate(){
        RedisTemplate<String, User> template = new RedisTemplate<>();
        template.setKeySerializer(new StringRedisSerializer());
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(new Jackson2JsonRedisSerializer<>(User.class));
        template.setConnectionFactory(jedisConnectionFactory());
        return template;
    }
}
```

6. 创建了一个封装 Redis 中数据操作的服务接口 `UserService`，并在其实现类中实现对用户数据的 CRUD：

```

@Service
public class UserServiceImpl implements UserService {
    private static final String USERKEY = "user";

    private HashOperations<String, Object, Object> operations;

    @Autowired
    private RedisTemplate<String, User> redisTemplate;

    @PostConstruct
    public void initOperations() { this.operations = redisTemplate.opsForHash(); }

    @Override
    public User save(User user) {
        this.operations.put(USERKEY, user.getId(), user);
        return user;
    }

    @Override
    public User findById(String id) { return (User) this.operations.get(USERKEY, id); }

    @Override
    public User update(User user) {
        save(user);
        return user;
    }

    @Override
    public void delete(String id) { this.operations.delete(USERKEY, id); }
}

```

7. 使用一个 Spring 的 ControllerRestController 创建操作用户的 RESTful 接口：

```

@RestController
@RequestMapping("/rest/user")
public class UserController {
    @Autowired
    private UserService userRepository;

    @PostMapping("/{id}")
    public User add(@PathVariable String id, @RequestParam String name,
        @RequestParam String sex, @RequestParam String nation){
        return userRepository.save(
            new User(id, name, sex, nation, Instant.now().getEpochSecond())
        );
    }

    @GetMapping("/{id}")
    public User findById(@PathVariable String id) { return userRepository.findById(id); }

    @PutMapping("/{id}")
    public User updateUserById(@PathVariable String id,
        @RequestParam String name, @RequestParam String sex,
        @RequestParam String nation, @RequestParam long register_time){
        return userRepository.update(
            new User(id, name, sex, nation, register_time)
        );
    }

    @DeleteMapping("/{id}")
    public void deleteUserById(@PathVariable String id) { userRepository.delete(id); }
}

```

8. 编码完成后，运行示例工程并使用 Swagger-ui（地址为<http://127.0.0.1:8080/swagger-ui.html>）

对 API 进行测试，如图4.4所示。

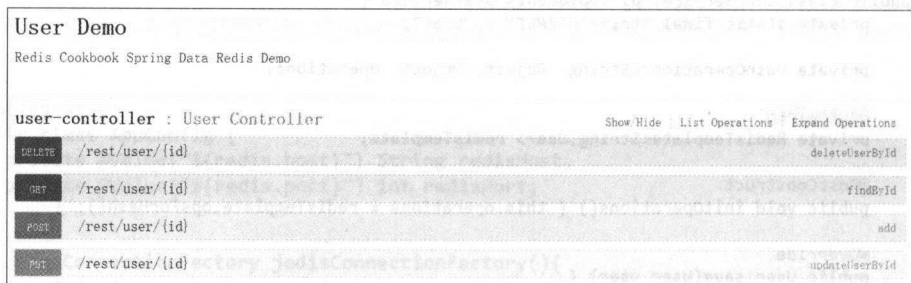


图 4.4 测试 API

9. 作为一个简单的测试,我们使用 POST 方法调用 /rest/user/{id} 来创建一个用户,如图4.5所示。

Parameters				
Parameter	Value	Description	Parameter Type	Data Type
id	0000088211	id	path	string
name	Mike	name	query	string
sex	Male	sex	query	string
nation	US	nation	query	string

图 4.5 创建用户

10. 在创建了一个名为 Mike 的用户后，我们使用 redis-cli 查看用户的数据：

```
127.0.0.1:6379> hgetall user
1) "0000088211"
2) {"id":"0000088211","name":"Mike","sex":"Male","nation":"US",
  "register_time":1507448493}
```

4.7.3 工作原理

对于 Spring Data Redis 而言，我们应该创建的第一个实例是 RedisTemplate，这个实例将被用于 Redis 的数据操作。在实例化 RedisTemplate 前，需要通过 JedisConnectionFactory 设置 Redis 服务器的 IP 地址和端口。由于需要更可靠的连接管理，我们还启用了 Redis 客户端的连接池。

在获取了 RedisTemplate 实例后，我们分别设置键、哈希键和哈希值的 Serializer。在本例中，对于键及哈希数据的键，我们只使用了纯文本。而对于哈希值，为了更好地组织数据，我们使用了 JSON 格式。

由于我们已经准备好了 `RedisTemplate` 实例及其相关的设置，所以可以在 Spring 服务中使用该实例实现 Redis 中用户数据的 CRUD 操作。

在最后的步骤中，我们创建了 RESTful API 作为操作用户数据的接口。

在测试阶段，我们使用 Swagger 将一个用户的数据通过 REST API 发送给服务器，并使用 `redis-cli` 对 Redis 中的数据进行验证。

4.7.4 相关内容

- 有关 Spring Redis Data 的更多细节，请参阅其主页：<https://projects.spring.io/spring-data-redis/>。
- 在测试阶段，我们使用了一款流行的 Web 开发工具 Swagger。如果读者对其感兴趣，可以在如下链接中找到更多细节：<https://swagger.io/>。

4.8 使用 Redis 编写 MapReduce 作业

如果读者是一名大数据工程师，那么 Redis 可能会在您的应用程序设计和开发中发挥重要的作用。在批处理作业场景中，我们可以在 Redis 中存储数据来实现以分布式的方式执行一些复杂的计算算法。对于在线查询而言，我们可以将结果数据集缓存在 Redis 服务器上来实现更好的性能。

在本章的最后两个案例中，我们将向读者展示如何使用在大数据世界中非常流行的两个分布式计算框架 MapReduce 和 Spark 操作 Redis 中的数据。

4.8.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。在学习这一节的内容之前，我们还需要使用 `FLUSHALL` 命令清除 Redis 实例中的所有数据。

对 IDE 和 JDK 的要求与前一个案例使用 *Spring Data* 连接到 Redis 相同。

我们推荐在 Hadoop 集群上运行本案例的 MapReduce 作业，但这也不是必须的。出于演示目的，我们可以在一个本地环境中调试和运行 MapReduce 作业作为替代。

学习本案例需要对 MapReduce 有基本的了解。

4.8.2 操作步骤

下面，让我们学习如何使用 Redis 编写 MapReduce 作业。假定 *Relp* 中的每个用户都有一定可用于支付服务费用的信用额度；出于促销的目的，我们要为 *Relp* 中的每个用户增加 10 美元的信用余额以鼓励用户使用 *Relp*。我们希望使用 MapReduce 框架以分布式的方式实现上述的需求。

1. 首先，使用如下的 shell 脚本 preparedata_mr.sh 准备测试数据：

```
$ bash preparedata_mr.sh
```

```
OK
```

```
unix2dos: converting file mr.data to DOS format ...
```

```
All data transferred. Waiting for the last reply...
```

```
Last reply received from server.
```

```
errors: 0, replies: 10000
```

2. 我们可以使用 redis-cli 来浏览数据：

```
127.0.0.1:6379> SCAN 0
```

```
1) "7"
```

```
2) 1) "user:00000006"
```

```
2) "user:00000000"
```

```
3) "user:00000008"
```

```
4) "user:00000004"
```

```
5) "user:00000003"
```

```
6) "user:00000002"
```

```
7) "user:00000007"
```

```
8) "user:00000001"
```

```
9) "user:00000009"
```

```
10) "user:00000005"
```

```
127.0.0.1:6379> SCAN 7
```

```
1) "0"
```

```
2) 1) "user:00000010"
```

3. 打开 IDEA，然后创建一个使用 Maven 进行依赖管理的新项目。

4. 点击 Next 继续，并填入项目的元数据。

5. 在 pom.xml 中添加 MapReduce 和其他的相关依赖：

```
<dependency>
```

```
  <groupId>org.apache.hadoop</groupId>
```

```
  <artifactId>hadoop-client</artifactId>
```

```
  <version>2.6.5</version>
```

```
</dependency>
```

6. 之后，为用户对象创建一个 Model 类，User，作为 JSON 数据的 POJO 对象。

7. 给 MapReduce 作业定制 InputFormat，以便从 Redis 中获取数据：

```

public class RedisHashInputFormat extends InputFormat<Text, Text> {
    private static final int IDLENGTH = 10;
    public static final String REDIS_HOST_CONF = "mr.redishashinputformat.host";
    public static final String REDIS_HASH_PREFIX_CONF = "mr.redishashinputformat.hashprefix";
    public static final String REDIS_BEGIN_CONF = "mr.redishashinputformat.begin";
    public static final String REDIS_END_CONF = "mr.redishashinputformat.end";
    public static final String REDIS_PLENGTH_CONF = "mr.redishashinputformat.plength";

    public List<InputSplit> getSplits(JobContext jobContext) throws IOException, InterruptedException {
        val host = jobContext.configuration.get(REDIS_HOST_CONF);
        val hashPrefix = jobContext.configuration.get(REDIS_HASH_PREFIX_CONF);
        val begin = Integer.parseInt(jobContext.configuration.get(REDIS_BEGIN_CONF));
        val end = Integer.parseInt(jobContext.configuration.get(REDIS_END_CONF));
        val plength = Integer.parseInt(jobContext.configuration.get(REDIS_PLENGTH_CONF));

        // Create an input split for each host
        val splits = [];
        var initKey = "";
        for (val i : [begin, end]){
            val number = StringUtils.leftPad(i, IDLENGTH, padChar: '0');
            val key = number[:IDLENGTH-plength];
            if(initKey != key){
                splits += new RedisHashInputSplit(host, hashPrefix, key);
                initKey = key;
            }
        }

        Log.info("Input splits to process: ${splits.size()}");
        return splits;
    }
}

```

8. 对于哈希键的每个分区，我们使用自定义的 `InputSplit` 创建输入分片来获取数据：

```

public class RedisHashInputSplit extends InputSplit implements Writable {
    private String host;
    private String prefix;
    private String key;

    public void write(DataOutput out) throws IOException {
        out.writeUTF(host);
        out.writeUTF(prefix);
        out.writeUTF(key);
    }

    public void readFields(DataInput in) throws IOException {
        this.host = in.readUTF();
        this.prefix = in.readUTF();
        this.key = in.readUTF();
    }

    public long getLength() throws IOException, InterruptedException {
        return 0;
    }

    public String[] getLocations() throws IOException, InterruptedException {
        return [host];
    }
}

```

9. 在每一个分片中，继承 `RecordReader`，使用 `Jedis` 迭代 Redis 中的数据：


```

public class RedisHashRecordReader extends RecordReader<Text, Text> {
    private Iterator<Map.Entry<String, String>> keyValueMapIter = null;
    private Text rrKey = new Text(), rrValue = new Text();
    private float processedKVs = 0, totalKVs = 0;
    private Map.Entry<String, String> currentEntry = null;
    private String prefix, host, key;
    private Jedis jedis;

    public void initialize(InputSplit split, TaskAttemptContext taskAttemptContext)
        throws IOException, InterruptedException {
        host = split.locations.first();
        prefix = split.prefix;
        key = split.key;
        String hashKey = prefix+"."+key;

        jedis = new Jedis(host);
        log.info("Connect to $host");
        jedis.connect();
        jedis.client.setTimeoutInfinite();

        totalKVs = jedis.hlen(hashKey);
        keyValueMapIter = jedis.hgetAll(hashKey).entrySet().iterator();
    }

    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (keyValueMapIter.hasNext()) {
            currentEntry = keyValueMapIter.next();
            rrKey.set(key+currentEntry.key);
            rrValue.set(currentEntry.value);
            return true;
        }
    }
}

```

10. 在获取数据之后，设置输出映射将每个用户的信用余额增加 10 美元：

```

public class RedisOutputMapper extends Mapper<Object, Text, Text, Text> {
    public static final String REDIS_BALANCE_CONF = "mr.redishashinputformat.balnce";
    private Text outkey = new Text();
    private Text outvalue = new Text();

    @Override
    protected void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        val addBalance = Long.parseLong(context.configuration.get(REDIS_BALANCE_CONF));
        val mapper = new ObjectMapper();
        val user = mapper.readValue(value.toString(), User.class);

        user.balance = addBalance+user.balance;
        // Set our output key and values
        outkey.set(key);
        outvalue.set(mapper.writeValueAsString(user));

        context.write(outkey, outvalue);
    }

    public static void setBalance(Job job, String balance) {
        job.configuration.set(REDIS_BALANCE_CONF, balance);
    }
}

```

11. 继承 OutputFormat 和 RecordWriter，将数据写回 Redis 服务器：

```

public class RedisHashRecordWriter extends RecordWriter<Text, Text> {
    private static final int IDLENGTH = 10;
    private final int pLength;
    private final String prefix;
    private Jedis jedis;

    public RedisHashRecordWriter(
        String host, String pLength, String prefix) {
        this.pLength = Integer.parseInt(pLength);
        this.prefix = prefix;
        jedis = new Jedis(host);
        jedis.connect();
    }

    public void write(Text key, Text value)
        throws IOException, InterruptedException {
        String key1 = key.toString().substring(0, IDLENGTH - pLength);
        String key2 = key.toString().substring(IDLENGTH - pLength);
        jedis.hset( key: prefix + ":" + key1, key2, value.toString());
    }

    public void close(TaskAttemptContext taskAttemptContext)
        throws IOException, InterruptedException {
        jedis.close();
    }
}

```

12. 最后，创建包含 `main()` 函数的类。

```

job.jarByClass = Application.class;
job.setMapperClass(RedisOutputMapper.class);

RedisOutputMapper.setBalance(job, balance);

job.inputFormatClass = RedisHashInputFormat.class;
RedisHashInputFormat.setRedisHost(job, host);
RedisHashInputFormat.setHashPrefix(job, hashPrefix);
RedisHashInputFormat.setBegin(job, begin);
RedisHashInputFormat.setEnd(job, end);
RedisHashInputFormat.setPLength(job, pLength);

job.outputFormatClass = RedisHashOutputFormat.class;
RedisHashOutputFormat.setRedisHost(job, host);
RedisHashOutputFormat.setPLength(job, pLength);

job.outputKeyClass = Text.class;
job.outputValueClass = Text.class;

//Wait for job completion
return (job.waitForCompletion( verbose: true) ? 0 : 1);

```

13. 提交作业：

```
...
2017-10-10 11:54:42,154 INFO [main] mapreduce.Job (Job.java:
    monitorAndPrintJob(1374)) - map 100% reduce 0%
2017-10-10 11:54:42,154 INFO [main] mapreduce.Job (Job.java:
    monitorAndPrintJob(1385)) - Job job_local736372500_0001 completed
    successfully
2017-10-10 11:54:42,184 INFO [main] mapreduce.Job (Job.java:
    monitorAndPrintJob(1392)) - Counters: 18
...
```

14. 在 redis-cli 中使用 hget 命令获取哈希值来检查作业是否完成：

```
127.0.0.1:6379> hget "user:0000001" 123
"{\"name\":\"Jack\\",\"sex\":\"m\\",\"rtime\":\"1507607748117668688\",\"nation\":\"\\
uk\\",\"balance\":\"103\"}"
```

4.8.3 工作原理

为了生成一些用于演示的数据，我们通过脚本将用户的测试数据导入 Redis。然后，按用户 ID 的前 7 个字符对示例数据进行了分区，并将其存储为哈希结构。

鉴于 MapReduce 应用中测试数据的特性，我们首先通过继承 InputFormat 类定制了一个 InputFormat，用于为每个用户数据的分区创建 InputSplit。一个 InputSplit 对应一个哈希键。

对于每个分片，我们使用 RedisHashRecordReader (RecordReader 的子类) 获取相应哈希键的数据。在 RecordReader 初始化的过程中，我们使用 Jedis 连接到 Redis 服务器，并调用 HLEN 命令获取哈希键的长度，然后使用 HGETALL 命令获取哈希键的所有数据。之后，相当于提供一个迭代器。

我们使用一个 mapper 来增加每个用户的信用余额。最后，我们使用定制的 OutputFormat 和 RecordWriter 按照原始的分区规则将处理后的用户数据写回 Redis。

编码完成后，我们提交 MapReduce 作业并检查结果。所有用户数据的信用余额都按预期地增加了 10 美元。

4.8.4 相关内容

- 在本地运行 MapReduce 作业的方法超出本书的讨论范围；请参阅如下链接中的指南：
<https://goo.gl/3VvYwA>。

4.9 使用 Redis 编写 Spark 作业

在前面的案例中，我们讨论了如何编写 MapReduce 作业来读写 Redis 中的数据。随着大规模计算的发展，近年来 Apache Spark 比 MapReduce 得到了更多的关注。Apache Spark 是一个开源的分布式大数据计算引擎。与 MapReduce 相比，它提供了更好的性能以及更强大、对用户更友好的 API。

为了能在 Spark 中更方便地使用 Redis，Redis Labs 提供了一个用于操作 Redis 中数据的 connector。在本案例中，我们将展示如何使用 Spark-Redis connector 库来读写 Redis 中的数据。

4.9.1 准备工作

我们需要按照启动和停止 Redis 一节中的步骤安装一个 Redis 服务器，并使用 `redis-cli` 连接到这个 Redis 服务器。在学习这一节的内容之前，我们还需要使用 `FLUSHALL` 命令清除 Redis 实例中的所有数据。

对 IDE 和 JDK 的要求与此前的案例使用 *Spring Data* 连接到 Redis 相同。在本书编写时，Spark-Redis 只能使用 Scala API，所以 Scala v2.11 和 IDEA 的 Scala 插件是必需的。此外，学习本案例还需要对 Spark 和 Scala 有基本的了解。

我们可以将作业提交到单独的 Spark 集群或 Yarn 集群中运行。出于演示的目的，我们将在本地模式中调试和运行 Spark 作业。

4.9.2 操作步骤

在前面的案例中，我们假定了每个用户都有信用余额。为了说明如何使用 Redis 编写 Spark 作业，我们将在本例中计算 Relp 中所有用户的余额之和：

1. 首先，使用如下的 shell 脚本 `preparedata_mr.sh` 准备测试数据：

```
$ bash preparedata_mr.sh
OK
unix2dos: converting file mr.data to DOS format ...
All data transferred. Waiting for the last reply...
Last reply received from server.
errors: 0, replies: 10000
```

2. 我们可以使用 `redis-cli` 来浏览数据：

```
127.0.0.1:6379> SCAN 0
1) "7"
2) 1) "user:0000006"
```

```
2) "user:0000000"
3) "user:0000008"
4) "user:0000004"
5) "user:0000003"
6) "user:0000002"
7) "user:0000007"
8) "user:0000001"
9) "user:0000009"
10) "user:0000005"
127.0.0.1:6379> SCAN 7
1) "0"
2) 1) "user:0000010"
```

3. 打开 IDEA，创建一个新的 Scala 项目，并填入项目的元数据。

4. 使用 Maven 对项目进行依赖管理。

5. 在 pom.xml 中添加 Spark-Redis 相关的仓库和依赖：

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
  <!-- https://mvnrepository.com/artifact/RedisLabs/spark-redis -->
  <dependency>
    <groupId>RedisLabs</groupId>
    <artifactId>spark-redis</artifactId>
    <version>0.3.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.1.0</version>
  </dependency>

  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.11</artifactId>
```

```

    <version>2.1.0</version>
</dependency>

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.1.0</version>
</dependency>

<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
</dependencies>

```

6. 之后，创建继承 App trait 的对象 SumBalance。

7. 在 SumBalance 的主体中，我们首先创建一个 SparkConf 类型的变量，并在其中设置运行模式和 Redis 服务器配置：

```

val conf = new SparkConf()
    .setMaster("local")
    .setAppName("Spark Redis Demo")
    .set("redis.host", "192.168.1.7")

```

8. 得到 SparkConf 类型的变量后，初始化 SparkSession，并得到 Spark-Redis 需要用到的 SparkContext：

```

//初始化sparksession
val sparkSession = SparkSession.builder.
    master("local")
    .config(conf)
    .appName("spark session example")
    .getOrCreate()

//获取spark-redis库所需要的sparkcontext
val sc = sparkSession.sparkContext

```

9. 使用 fromRedisKeyPattern 方法从 Redis 中读取哈希数据：


```
//从Redis中读取哈希数据
```

```
val userHashRDD= sc.fromRedisKeyPattern("user*").getHash()
```

10. 在计算余额的总和之前，先解析 JSON 数据：

```
import sparkSession.implicit._
```

```
val ds = sparkSession.createDataset(userHashRDD)
```

```
//准备JSON的模式
```

```
val schema = StructType(Seq(
  StructField("name", StringType, true),
  StructField("sex", StringType, true),
  StructField("rtime", LongType, true),
  StructField("nation", StringType, true),
  StructField("balance", DoubleType, true)
))
```

```
//解析JSON数据并将列重命名
```

```
val namedDS = ds
  .withColumnRenamed("_1", "id")
  .withColumnRenamed("_2", "jsondata")
  .withColumn("jsondata", from_json($"jsondata", schema))
```

11. 数据准备好后，我们就可以开始进行数学运算了。在计算完余额的总和之后，将结果作为一个简单的字符串类型的键值对写回 Redis 用于后续的在线查询：

```
//生成结果（[String, String] RDD）
```

```
val totalBalanceRDD = namedDS.agg(sum($"jsondata.balance")).rdd.map(total => ("totalBalance", total.get(0).toString()))
```

12. 最后，使用 Spark-Redis 提供的 toRedisKV 方法将结果写回 Redis：

```
//将结果写回Redis
```

```
sc.toRedisKV(totalBalanceRDD)
```

13. 提交作业，并测试是否按预期完成了求和：

```
...
```

```
17/10/10 21:02:28 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks
have all completed, from pool
```

```

17/10/10 21:02:28 INFO DAGScheduler: ResultStage 1 (foreachPartition at
redisFunctions.scala:226) finished in 0.130 s
17/10/10 21:02:28 INFO DAGScheduler: Job 0 finished: foreachPartition at
redisFunctions.scala:226, took 2.675819 s
17/10/10 21:02:28 INFO SparkContext: Invoking stop() from shutdown hook
17/10/10 21:02:28 INFO SparkUI: Stopped Spark web UI at http
://192.168.56.1:4040
17/10/10 21:02:28 INFO MapOutputTrackerMasterEndpoint:
MapOutputTrackerMasterEndpoint stopped!
17/10/10 21:02:28 INFO MemoryStore: MemoryStore cleared
17/10/10 21:02:28 INFO BlockManager: BlockManager stopped
17/10/10 21:02:28 INFO BlockManagerMaster: BlockManagerMaster stopped
17/10/10 21:02:28 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint
: OutputCommitCoordinator stopped!
17/10/10 21:02:28 INFO SparkContext: Successfully stopped SparkContext
17/10/10 21:02:28 INFO ShutdownHookManager: Shutdown hook called

```

14. 在 `redis-cli` 中，我们得到了所有用户的余额的总和：

```

127.0.0.1:6379> GET totalBalance
"492569.0"

```

4.9.3 工作原理

上述的代码非常容易理解。我们应该注意到的一点是，根据 Spark-Redis 库的要求，当想要把数据作为简单的键值对写入 Redis 时，需要使用 `RDD[(String, String)]`。StringRDD 的第一部分是要在 Redis 中设置的键。因此，在本例中，我们向 Redis 中设置数据前先要将结果转换一个 StringRDD（名为 `totalBalance` 的键）。不同的 Redis 数据类型需要使用 Spark-Redis 中不同类型的 RDD。

4.9.4 更多细节

除了操作单个 Redis 服务器外，Spark-Redis 还支持在应用程序中连接到多个 Redis 集群或实例。

4.9.5 相关内容

- 有关 Spark-Redis 提供的更多 API，请参阅：<https://github.com/RedisLabs/spark-redis>。

第 5 章

复制

在本章中，我们将学习下列案例：

- 配置 Redis 的复制机制。
- 复制机制的调优。
- 复制机制的故障诊断。

5.1 本章概要

到目前为止，我们已经学习了如何设置和连接单个 Redis 服务器。在生产环境中，单个数据库实例常常存在诸如系统崩溃、网络连接闪断或突然断电等单点故障的问题。与其他大多数数据库系统一样，Redis 也提供了一个复制机制，使得数据能够从一个 Redis 服务器（master，主实例）复制到一个或多个其他的 Redis 服务器中（slave，从实例）。

复制不仅提高了整个系统的容错能力，还可以用于对系统进行水平扩展。在一个重读取的应用中，我们可以通过增加多个 Redis 只读从实例来减轻主实例的压力。

Redis 的复制机制是 Redis Cluster 的基础，而 Redis Cluster 在此基础上提供了高可用性。在本章中，我们将首先介绍 Redis 主从复制机制的工作原理，以及如何配置一个 Redis 主-从复制环境。之后，我们将展示关于 Redis 主从复制机制的调优指南和基本的故障诊断手段。

5.2 配置 Redis 的复制机制

在第 1 章开始使用 Redis 中，我们学习了如何配置 Redis 服务器。在默认情况下，Redis 服务器是以主实例模式运行的。在本案例中，我们将演示如何配置 Redis 的主从同步。

5.2.1 准备工作

我们需要按照第 1 章 开始使用 *Redis* 中下载和安装 *Redis* 的案例中描述的步骤安装一个 *Redis* 服务器。

首先，为 *Redis* 从实例准备一个配置文件。我们可以将 `redis.conf` 复制一份并重新命名为 `redis-slave.conf`，然后进行以下的修改：

```
port 6380
pidfile /var/run/redis_6380.pid
dir ./slave
slaveof 127.0.0.1 6379
```

不要忘记创建 `/redis/slave` 目录（如果不存在的话）：

```
$mkdir -p /redis/slave
```

5.2.2 操作步骤

接下来，让我们按照以下的步骤来学习如何配置 *Redis* 的主从复制机制。

1. 在本机上启动一个监听 6379 端口的 *Redis* 实例。如果实例已经在运行了，那么可以跳过这一步。我们将这个实例作为主实例：

```
$cd /redis
$bin/redis-server conf/redis.conf
```

2. 使用配置文件 `redis-slave.conf` 启动另一个 *Redis* 实例。我们将这个实例作为从实例：

```
$bin/redis-server conf/redis-slave.conf
```

3. 使用 `redis-cli` 打开两个终端，并分别连接到主实例 `127.0.0.1 :6379` 和从实例 `127.0.0.1:6380`：

```
$ bin/redis-cli -p 6379
127.0.0.1:6379>

$ bin/redis-cli -p 6380
127.0.0.1:6380>
```

4. 在两个终端上运行同时执行 `INFO REPLICATION`：

```
127.0.0.1:6379> INFO REPLICATION
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=557,lag=1
master_replid:1fee079fc47716706a59225779c56b0e7033f3b1
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:557
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:557
```

```
127.0.0.1:6380> INFO REPLICATION
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:7
master_sync_in_progress:0
slave_repl_offset:557
slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:1fee079fc47716706a59225779c56b0e7033f3b1
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:557
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:557
```

5. 在主实例上, 创建一个新的键:

```
127.0.0.1:6379> SET "new_key" "value"
```

OK

6. 在从实例上，尝试获取新键的值：

```
127.0.0.1:6380> GET "new_key"
"value"
```

7. 在从实例上，尝试创建一个新的键：

```
127.0.0.1:6380> SET "new_key_2" "value2"
(error) READONLY You can't write against a read only slave.
```

8. 关闭从实例：

```
127.0.0.1:6380> shutdown save
```

9. 在主实例上，创建另一个新的键：

```
127.0.0.1:6379> SET "another_new_key" "another_value"
OK
```

10. 重启从实例：

```
$bin/redis-server bin/redis-slave.conf
```

11. 检查从实例上是否存在 another_new_key：

```
127.0.0.1:6380> GET "another_new_key"
"another_value"
```

5.2.3 工作原理

在准备工作小节中，我们更新了配置文件，让从实例监听 6380 端口，同时使用了与主实例不同的工作目录。我们在 redis-slave.conf 中添加的 slave of 127.0.0.1 6379 表示监听 6380 端口的实例是 127.0.0.1:6379 的一个从实例。此外，SLAVEOF 同时也是可以在 redis-cli 中直接运行的命令，可以动态地将当前的 Redis 实例变为另一个实例的从实例。

INFO 命令输出了当前服务器的信息。在第四步中，我们使用了 INFO REPLICATION 命令来检查是否成功地建立了复制关系。大多数有关复制机制信息的含义都是直截了当的。master_replid 是在主实例启动时生成的随机字符串。在 Redis 的复制机制中，我们用 master_replid 来标识主实例。master_repl_offset 是复制流中的一个偏移标记，会随着主实例上数据事件的发生而增长。(master_replid; master_repl_offset) 对可被用来标识主实例复制流中的位置。

当主从实例之间网络连接通畅且建立了复制关系后，主实例会把将其接收到的写入命令转发给从实例执行，以实现主从实例之间的数据同步。

当从实例第一次与主实例连接时会发生什么呢？从实例在网络连接中断后会重新连接到主实例么？在 Redis 的复制机制中，共有两种重新同步机制：部分重新同步和完全重新同步。当一个 Redis 的从实例启动并连接到主实例时，从实例总是会尝试通过发送 `(master_replid; master_repl_offset)` 请求进行部分重新同步。其中，`(master_replid; master_repl_offset)` 表示与主实例同步的最后一个快照。如果主实例接受部分重新同步的请求，那么它会将从实例停止时的最后一个偏移处开始增量地进行命令同步。否则，则需要进行完全重新同步。当从实例第一次连接到它的主实例时，总是需要进行完全重新同步。我们将在后续复制机制的调优案例中对有关主实例如何决定是否接受部分重新同步请求的细节进行讨论。在进行完全重新同步时，为了将所有数据复制到从实例中，主实例需要将数据转储到一个 RDB 文件中，然后将这个文件发送给从实例。从实例接收到 RDB 文件后，会先将内存中的所有数据清除，再将 RDB 文件中的数据导入。主实例上的复制过程是完全异步的，因此并不会阻塞服务器处理客户端的请求。

显然，与完全重新同步相比，部分重新同步不需要从主实例中传输完整的数据转储文件。另外，将数据转储到 RDB 文件中会创建一个后台进程，还会有内存开销；我们将在下一章中对此进行学习。部分重新同步和完全重新同步的过程如图 5.1 所示。让我们看一看当从实例第一次连接到主实例时的日志。正如我们所看到的，由于从实例从来没有连接过主实例，也就没有 `(master_replid; master_repl_offset)`，因此它请求主实例进行了一次完全重新同步：

```
15516:S 15 Oct 15:30:15.412 * Connecting to MASTER 127.0.0.1:6379
15516:S 15 Oct 15:30:15.412 * MASTER <-> SLAVE sync started
15516:S 15 Oct 15:30:15.412 * Non blocking connect for SYNC fired the event.
15516:S 15 Oct 15:30:15.412 * Master replied to PING, replication can continue
...
15516:S 15 Oct 15:30:15.412 * Partial resynchronization not possible (no
cached master)
15516:S 15 Oct 15:30:15.421 * Full resync from master: 1
fee079fc47716706a59225779c56b0e7033f3b1:0
15516:S 15 Oct 15:30:15.511 * MASTER <-> SLAVE sync: receiving 175 bytes from
master
15516:S 15 Oct 15:30:15.511 * MASTER <-> SLAVE sync: Flushing old data
15516:S 15 Oct 15:30:15.511 * MASTER <-> SLAVE sync: Loading DB in memory
15516:S 15 Oct 15:30:15.512 * MASTER <-> SLAVE sync: Finished with success
```

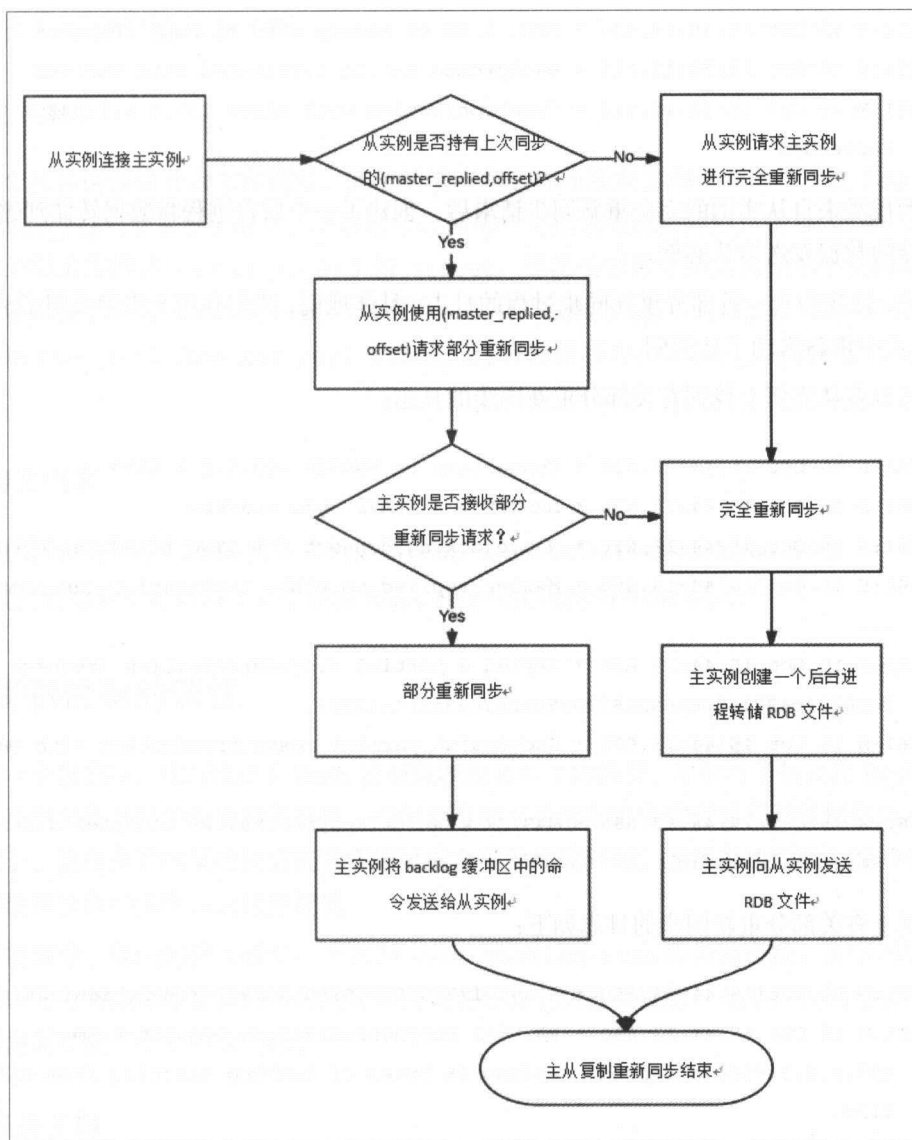


图 5.1 部分重新同步和完全重新同步的过程

下面是主实例的日志：

```

15511:M 15 Oct 15:30:15.412 * Slave 127.0.0.1:6380 asks for synchronization
15511:M 15 Oct 15:30:15.412 * Full resync requested by slave 127.0.0.1:6380
15511:M 15 Oct 15:30:15.413 * Starting BGSAVE for SYNC with target: disk
15511:M 15 Oct 15:30:15.413 * Background saving started by pid 15520
15520:C 15 Oct 15:30:15.423 * DB saved on disk
  
```

```
15520:C 15 Oct 15:30:15.424 * RDB: 6 MB of memory used by copy-on-write
15511:M 15 Oct 15:30:15.511 * Background saving terminated with success
15511:M 15 Oct 15:30:15.511 * Synchronization with slave 127.0.0.1:6380
succeeded
```

主实例接受来自从实例的完全重新同步请求后，创建了一个后台进程将数据转储到磁盘上。然后，将转储的数据发送给从实例。

接下来，让我们看一看部分重新同步过程的日志。具体地说，我们在第 8 步中关闭了从实例，然后在第 10 步中重新启动了从实例。

我们可以在从实例上找到有关部分重新同步的日志：

```
15561:S 15 Oct 15:44:13.650 * Connecting to MASTER 127.0.0.1:6379
15561:S 15 Oct 15:44:13.650 * MASTER <-> SLAVE sync started
15561:S 15 Oct 15:44:13.650 * Non blocking connect for SYNC fired the event.
15561:S 15 Oct 15:44:13.650 * Master replied to PING, replication can continue
...
15561:S 15 Oct 15:44:13.650 * Trying a partial resynchronization (request 1
fee079fc47716706a59225779c56b0e7033f3b1:1126).
15561:S 15 Oct 15:44:13.650 * Successful partial resynchronization with master
.
15561:S 15 Oct 15:44:13.650 * MASTER <-> SLAVE sync: Master accepted a Partial
Resynchronization.
```

主实例上有关部分重新同步的日志如下：

```
15511:M 15 Oct 15:44:13.650 * Slave 127.0.0.1:6380 asks for synchronization
15511:M 15 Oct 15:44:13.650 * Partial resynchronization request from
127.0.0.1:6380 accepted. Sending 55 bytes of backlog starting from offset
1126.
```

在前面的服务器日志中，我们可以看到从实例使用 (master_replid;offset) 对 (1fee079fc47716706a59225779c56b0e7033f3b1; 1126) 发送了一个部分的重新同步请求。主实例接受了这个请求，并开始从 backlog 缓冲区中发送位于 1126 偏移处的命令。请注意，从实例重启后进行部分重新同步是 Redis 4.0 中的一个新特性。在 Redis 4.0 的实现中，master_replid 和 offset 存储在 RDB 文件中。当从实例被优雅地关闭并重新启动时，Redis 能够从 RDB 文件中重新加载 master_replid 和 offset，从而使部分重新同步成为可能。

在第 7 步中，我们尝试在从实例上创建一个新键，但是得到一个错误（服务器处于只读模式）。这是因为我们在配置文件中配置了 slave-read-only yes。在大多数情况下，最好保持这种配置，

以避免主从实例之间的数据不一致。

5.2.4 更多细节

当一个从实例被提升为主实例时，其他的从实例必须与新主实例重新同步。在 Redis 4.0 之前，因为 `master_replid` 发生了改变，所以这个过程是一个完全的重新同步。在 Redis 4.0 之后，新主实例会记录旧主实例的 `master_replid` 和 `offset`，因此能够接受来自其他从实例的部分重新同步请求（即使请求中的 `master_replid` 不同）。更具体地说，当主实例发生故障迁移时，在新的主实例上，`(master_replid;master_repl_offset+1)` 将被复制为 `(master_replid2;second_repl_offset)`。

5.2.5 相关内容

- 有关复制机制的 Redis 官方文档，请参阅：<https://redis.io/topics/replication>。
- 我们将会在第 6 章持久化中解释 Redis 的持久化机制和 RDB 格式。

5.3 复制机制的调优

在上一个案例中，我们讨论了 Redis 复制机制的基本工作流程，并学习了如何在 Redis 中配置主从复制。虽然主从复制的配置非常简单，但仍然有很多关键配置参数需要我们特别注意。此外，在某些情况下，这些参数的某些缺省配置甚至可能会导致性能问题。在本章的后两个案例中，我们将从调优和故障诊断的角度讨论这些配置。

在本案例中，我们将深入研究一个名为 `repl-backlog-size` 的关键参数，并学习如何通过调整该参数以充分利用部分重新同步的优势来实现更好的主从复制性能。此外，我们还会讨论一些其他可能对复制性能产生影响的参数。

5.3.1 准备工作

我们需要按照第 1 章开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。

在本案例中，出于演示的目的，我们使用 iptables 作为防火墙，以模拟网络连接中断的情形；在操作 iptables 时需要有 root 权限。

5.3.2 操作步骤

接下来，让我们按照以下的步骤来学习如何进行 Redis 复制机制的调优。

1. 打开终端并将当前用户切换为 root 用户，然后通过控制台设置有关主从实例基本信息的变量：

```
$ M_IP=127.0.0.1
$ M_PORT=6379
$ M_RDB_NAME=master.rdb
$ M_OUT=master.out
$ S1_IP=127.0.0.2
$ S1_PORT=6380
$ S1_OUT=slave_1.out
$ S1_RDB_NAME=slave_1.rdb
```

2. 启动两个 Redis 实例：

```
$ nohup /redis/bin/redis-server --port $M_PORT --bind $M_IP --dbfilename
    $M_RDB_NAME > $M_OUT &
$ nohup /redis/bin/redis-server --port $S1_PORT --bind $S1_IP --dbfilename
    $S1_RDB_NAME > $S1_OUT&
```

3. 等待一段时间，让 Redis 实例完成启动。等待结束后，向一个 Redis 实例中设置一些测试数据：

```
$ sleep 10
$ echo set redis hello | nc $M_IP $M_PORT
$ echo lpush num 1 2 3 | nc $M_IP $M_PORT
```

4. 通过调用 SLAVEOF 命令配置这两个 Redis 实例之间的主从复制。完成主从复制的配置后等待 10 秒钟：

```
$ echo slaveof 127.0.0.1 6379 | nc $S1_IP $S1_PORT
$ sleep 10
```

5. 数据同步完成之后，使用 iptables 中断主实例和从实例之间的网络连接：

```
$ echo "modify iptables"
$ iptables -I INPUT -s 127.0.0.1 -d 127.0.0.2 -j DROP
$ iptables -I OUTPUT -s 127.0.0.2 -d 127.0.0.1 -j DROP
```

6. 在网络连接中断期间，向主实例中导入一些测试数据：

```
$ bash preparerepldata.sh 1000
```

```
$ du -sh repl.data
$ cat repl.data | /redis/bin/redis-cli --pipe -h $M_IP -p $M_PORT
```

7. 然后，通过调用 `iptables` 命令恢复主实例和从实例之间的网络连接。在恢复网络连接之前，等待 70 秒：

```
$ sleep 70
$ echo "restore iptables"
$ iptables -D INPUT -s 127.0.0.1 -d 127.0.0.2 -j DROP
$ iptables -D OUTPUT -s 127.0.0.2 -d 127.0.0.1 -j DROP
```

8. 网络重新连接后，等待 10 秒让主从实例进行数据重新同步，然后同时关闭主实例和从实例：

```
$ sleep 10
$ echo "SHUTDOWN" | nc $M_IP $M_PORT
$ echo "SHUTDOWN" | nc $S1_IP $S1_PORT
```

9. 检查主实例和从实例的日志。我们可以找到以下与主从复制相关的内容：

```
# cat master.out
...
29086:M 15 Oct 20:25:49.020 # Connection with slave 127.0.0.2:6380 lost.
29086:M 15 Oct 20:26:04.037 * Slave 127.0.0.2:6380 asks for synchronization
29086:M 15 Oct 20:26:04.037 * Partial resynchronization request from
127.0.0.2:6380 accepted. Sending 139107 bytes of backlog starting from
offset 15.
...

# cat slave_1.out
...
29087:S 15 Oct 20:25:49.020 # MASTER timeout: no data nor PING received...
29087:S 15 Oct 20:25:49.020 # Connection with master lost.
29087:S 15 Oct 20:25:49.020 * Caching the disconnected master state.
29087:S 15 Oct 20:25:49.020 * Connecting to MASTER 127.0.0.1:6379
29087:S 15 Oct 20:25:49.020 * MASTER <-> SLAVE sync started
29087:S 15 Oct 20:26:04.037 * Non blocking connect for SYNC fired the event.
29087:S 15 Oct 20:26:04.037 * Master replied to PING, replication can continue
...
29087:S 15 Oct 20:26:04.037 * Trying a partial resynchronization (request
e39b33ba4bb4bb0bf3623ad09d385a856f27463c:15).
```



```
29087:S 15 Oct 20:26:04.037 * Successful partial resynchronization with master
```

```
29087:S 15 Oct 20:26:04.037 * MASTER <-> SLAVE sync: Master accepted a Partial
Resynchronization.
```

```
...
```

10. 删除所有日志和转储文件。重复前面的步骤。这一次，我们增大在第 6 步网络断开期间导入到主实例中的数据量：

```
$ bash preparerepldata.sh 11000
```

11. 再次检查主实例和从实例的日志：

```
# cat master.out
```

```
...
```

```
31156:M 15 Oct 20:31:01.747 # Disconnecting timedout slave: 127.0.0.2:6380
```

```
31156:M 15 Oct 20:31:01.747 # Connection with slave 127.0.0.2:6380 lost.
```

```
31156:M 15 Oct 20:31:32.809 * Slave 127.0.0.2:6380 asks for synchronization
```

```
31156:M 15 Oct 20:31:32.809 * Unable to partial resync with slave
```

```
127.0.0.2:6380 for lack of backlog (Slave request was: 15).
```

```
31156:M 15 Oct 20:31:32.809 * Starting BGSAVE for SYNC with target: disk
```

```
31156:M 15 Oct 20:31:32.810 * Background saving started by pid 21293
```

```
21293:C 15 Oct 20:31:32.864 * DB saved on disk
```

```
21293:C 15 Oct 20:31:32.864 * RDB: 8 MB of memory used by copy-on-write
```

```
31156:M 15 Oct 20:31:32.898 * Background saving terminated with success
```

```
31156:M 15 Oct 20:31:32.898 * Synchronization with slave 127.0.0.2:6380
succeeded
```

```
...
```

```
# cat slave_1.out
```

```
...
```

```
31157:S 15 Oct 20:31:01.746 # MASTER timeout: no data nor PING received...
```

```
31157:S 15 Oct 20:31:01.746 # Connection with master lost.
```

```
31157:S 15 Oct 20:31:01.746 * Caching the disconnected master state.
```

```
31157:S 15 Oct 20:31:01.746 * Connecting to MASTER 127.0.0.1:6379
```

```
31157:S 15 Oct 20:31:01.746 * MASTER <-> SLAVE sync started
```

```
31157:S 15 Oct 20:31:32.809 * Non blocking connect for SYNC fired the event.
```

```
31157:S 15 Oct 20:31:32.809 * Master replied to PING, replication can continue
```

```
...
31157:S 15 Oct 20:31:32.809 * Trying a partial resynchronization (request
a866cd909b0ceb7bbed690f73f633f97a471fd3d:15).
31157:S 15 Oct 20:31:32.810 * Full resync from master:
a866cd909b0ceb7bbed690f73f633f97a471fd3d:1529121
31157:S 15 Oct 20:31:32.810 * Discarding previously cached master state.
31157:S 15 Oct 20:31:32.898 * MASTER <-> SLAVE sync: receiving 957221 bytes
from master
31157:S 15 Oct 20:31:32.899 * MASTER <-> SLAVE sync: Flushing old data
31157:S 15 Oct 20:31:32.899 * MASTER <-> SLAVE sync: Loading DB in memory
31157:S 15 Oct 20:31:32.914 * MASTER <-> SLAVE sync: Finished with success
..
```

5.3.3 工作原理

在前一节中，我们启动了两个 Redis 服务器，并在它们之间配置了主从复制。通过检查主实例和从实例的日志，我们能够发现第一次复制是一次完整的数据同步。因为这是主实例和从实例之间的第一次同步数据，所以这并不奇怪：

```
# cat master.out
...
31156:M 15 Oct 20:29:40.618 * Ready to accept connections
31156:M 15 Oct 20:29:51.637 * Slave 127.0.0.2:6380 asks for synchronization
31156:M 15 Oct 20:29:51.637 * Partial resynchronization not accepted:
Replication ID mismatch (Slave asked for '709
e798b196d833e4b6ff34f1e1cf1a392aa81c4', my replication IDs are '9
dd60b163d9aad07426704bd00c8fcdc5e509bd8' and
'0000000000000000000000000000000000000000000000000000000000000000')
31156:M 15 Oct 20:29:51.638 * Starting BGSAVE for SYNC with target: disk
31156:M 15 Oct 20:29:51.638 * Background saving started by pid 31175
31175:C 15 Oct 20:29:51.661 * DB saved on disk
31175:C 15 Oct 20:29:51.661 * RDB: 6 MB of memory used by copy-on-write
31156:M 15 Oct 20:29:51.736 * Background saving terminated with success
31156:M 15 Oct 20:29:51.736 * Synchronization with slave 127.0.0.2:6380
succeeded
```

然后，我们利用 iptables 切断了网络连接。在网络连接断开期间，我们生成了一些测试数据并导入到了 Redis 主实例中。之后，我们恢复了网络连接，并检查这两个 Redis 实例的日志。

有趣的是，不同大小的测试数据引发了不同类型的数据重新同步。发生这种情况的原因是，在 Redis 主实例失去与从实例的网络连接期间，主实例上的一段内存（实际是一个环形缓冲区）会跟踪最近所有的写入命令。这个缓冲区实际上是一个固定长度的列表。

在 Redis 中，我们将这个缓冲区称为 **replication backlog**。Redis 使用这个 backlog 缓冲区来决定究竟是进行完全重新同步还是部分重新同步。更具体地说，在发出 `SLAVEOF` 命令后，从实例使用最后一个 `offset` 和最后一个主实例的 ID (`master_replid`) 向主实例发送一个部分重新同步请求。当主实例和从实例之间的连接建立后，主实例首先会检查请求中的 `master_replid` 是否与它自己的 `master_replid` 一致。然后，主实例会检查请求中的 `offset` 能否从 backlog 缓冲区中获取。如果 `offset` 位于 backlog 的范围内，那么就可以从中获得连接断开期间的所有写入命令，这也就意味着能够进行部分重新同步。否则，如果主实例在连接断开期间接收到的写入命令的数量超过了 backlog 缓冲区的容量，那么部分重新同步请求会被拒绝；此时，完全重新同步将被启动。

默认情况下，backlog 缓冲区的大小是 1MB，这个容量在连接断开期间只能容纳少量的写入命令。在第一次测试时，我们生成并向主实例导入了约 96KB 的数据。因此，当从实例再次连接到主实例并请求进行部分重新同步时，主实例能够从 backlog 缓冲区中获取连接断开期间的写入命令，并将其发送给从实例。最后，从实例以部分复制的方式追赶上了主实例。部分重新同步的好处已经在此前配置 Redis 的复制机制的案例中进行了描述。

在第二个测试场景中，我们在网络连接断开期间生成并向主实例导入了 1.1MB 的数据。导入数据的大小比 backlog 缓冲区的默认大小要大，因此在重新连接后毫无疑问地要执行完全重新同步。

我们可以从此前的讨论中得出这样一个结论：当主实例和从实例之间网络连接中断时，backlog 缓冲区的默认大小并不足以应对高写入流量的情况。在多数情况下，我们需要把这个参数调整为更高的值以满足需求。通过在峰值期间使用 `INFO` 命令计算 `master_repl_offset` 的变化量，我们可以估算 backlog 缓冲区的合适大小：

```
t*(master_repl_offset2- master_repl_offset1)/(t2-t1)
t is how long the disconnection may last in seconds.
```

我们也可以利用这个公式来估算主实例和从实例之间的网络流量。

一般来说，将这个值设置为比 RDB 快照大小还大的值是没有意义的。因为在部分重新同步和完全重新同步中传输数据的大小几乎是相同的，所以这样做无法利用部分重新同步的优点。

5.3.4 更多细节

有关 backlog 的另一个参数是 `repl-backlog-ttl`。该参数的含义是：如果所有的从实例与主实例的连接全部断开，那么主实例等待多久释放 backlog 占用的内存。这个参数的默认值是 3600s，这通常不会有问题，因为与 Redis 实例占用的总内存相比 backlog 缓冲区是非常小的。

除了 backlog 的大小之外，还有一些其他的配置可以在某些情况下进行调优以获得更好的性能。从网络传输的角度看，我们可以通过将参数 `repl-disable-tcp-nodelay` 设为 `yes` 来减少带宽的使用。如果设置为 `yes`，Redis 会尝试将几个小包合并成一个包，这在主实例和从实例位置相距较远时有些作用。但是，我们需要额外注意的是，这个选项可能造成约 40 毫秒的复制延迟。

从主实例的 I/O 和内存角度来看，我们可以通过使用无磁盘复制直接将 RDB 的内容发送给从实例而无需在磁盘上创建 RDB 文件。这种机制可以在 RDB 快照过程中节省大量的磁盘 I/O 和内存。如果 Redis 所在主机上的磁盘速度不快或内存使用率很高，而网络带宽又足够，那么我们可以考虑打开这个选项。我们可以通过把 `repl-diskless-sync` 设置为 `yes` 来将复制机制从默认的 `disk-backed` 切换为 `diskless`。但是，这个特性目前仍处于实验阶段；所以，在生产环境中使用无磁盘复制时要多加小心。

5.3.5 相关内容

- 有关在 Redis 中设计部分同步，请参阅：<http://antirez.com/news/31>。
- 有关在 Redis 中设计无磁盘的同步，请参阅：<http://antirez.com/news/81>。
- 有关 Redis 复制机制的细节，请参阅本章中配置 Redis 的复制机制的案例。
- 读者还可以从如下链接中找到本案例中所提到的配置项背后的含义：<http://download.redis.io/redis-stable/redis.conf>。
- 如果读者不熟悉 `iptables` 命令，请参阅其帮助页面：<http://ipset.netfilter.org/iptables.man.html>。

5.4 复制机制的故障诊断

在实际的生产环境中，我们在使用 Redis 的复制机制时可能会遇到许多问题并陷入困境。诸如磁盘 I/O、网络连接、数据集大小和长时间的阻塞操作等很多因素都可能成为复制机制失效的根本原因。

在本案例中，我们将通过几个复制机制失效的案例和解决方案来学习 Redis 复制机制失效时的应对方法。

5.4.1 准备工作

我们需要按照第 1 章 开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。我们还需要按照本章中配置 Redis 的复制机制的案例中描述的步骤完成主从复制的配置。

为了生成大量的测试数据，我们使用了在第 2 章数据类型中键管理的案例中所提到的 `fake2db`，将一些假数据导入到 Redis 中。鉴于硬件性能的不同，读者可能需要几个小时来完成数据的导入：

```
# fake2db --rows 3000000 --db redis
```

此外,出于测试的目的,我们引入另一个 Redis 测试数据生成器 `redis-random-data-generator`,来实时地将测试数据写入 Redis。我们可以按照如下的方式进行安装:

```
$ sudo apt-get install npm
$ npm i redis-random-data-generator
$ wget https://raw.githubusercontent.com/SaminOz/redis-random-data-generator/master/generator.js
```

5.4.2 操作步骤

接下来,让我们按照以下步骤学习在遇到复制机制相关的问题时如何进行故障诊断。

1. 打开一个终端并使用 `redis-cli` 获取主实例和从实例主从复制相关的信息:

```
$ bin/redis-cli -p 6379 INFO REPLICATION
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=6380,state=online,offset=1288,lag=1
master_replid:8f7b9821477006200651baef11d6af7451dede3d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:1288
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:1288
```

```
$ bin/redis-cli -p 6380 INFO REPLICATION
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
master_last_io_seconds_ago:1
master_sync_in_progress:0
slave_repl_offset:1302
```

```

slave_priority:100
slave_read_only:1
connected_slaves:0
master_replid:8f7b9821477006200651baef11d6af7451dede3d
master_replid2:0000000000000000000000000000000000000000
master_repl_offset:1302
second_repl_offset:-1
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:1
repl_backlog_histlen:1302

```

2. 在查看了主从复制的状态之后，使用 Redis 的 `debug sleep` 命令将主实例阻塞 80 秒：

```

$ date;bin/redis-cli -p 6379 debug sleep 80; date
Tue Oct 17 16:15:54 CST 2017
OK
Tue Oct 17 16:17:14 CST 2017

```

3. 查看从实例和主实例的日志：

```

Slave:
21894:S 17 Oct 16:16:52.823 # MASTER timeout: no data nor PING received...
21894:S 17 Oct 16:16:52.823 # Connection with master lost.
21894:S 17 Oct 16:16:52.823 * Caching the disconnected master state.
21894:S 17 Oct 16:16:52.823 * Connecting to MASTER 127.0.0.1:6379
21894:S 17 Oct 16:16:52.823 * MASTER <-> SLAVE sync started
21894:S 17 Oct 16:16:52.823 * Non blocking connect for SYNC fired the event.
21894:S 17 Oct 16:17:14.353 * Master replied to PING, replication can continue
...
21894:S 17 Oct 16:17:14.353 * Trying a partial resynchronization (request 8
f7b9821477006200651baef11d6af7451dede3d:565381621).
21894:S 17 Oct 16:17:14.353 * Successful partial resynchronization with master
.
21894:S 17 Oct 16:17:14.353 * MASTER <-> SLAVE sync: Master accepted a Partial
Resynchronization.

Master:
22024:M 17 Oct 16:17:14.353 # Connection with slave 127.0.0.1:6380 lost.

```



```
22024:M 17 Oct 16:17:14.353 * Slave 127.0.0.1:6380 asks for synchronization
22024:M 17 Oct 16:17:14.353 * Partial resynchronization request from
127.0.0.1:6380 accepted. Sending 0 bytes of backlog starting from offset
565381621.
```

4. 这一次，在重新同步完成后，我们马上把从实例挂起：

```
$ date;bin/redis-cli -p 6380 debug sleep 80; date
Tue Oct 17 20:46:43 CST 2017
OK
Tue Oct 17 20:48:03 CST 2017
```

5. 查看主实例和从实例的日志：

Master:

```
22024:M 17 Oct 20:47:43.709 # Disconnecting timedout slave: 127.0.0.1:6380
22024:M 17 Oct 20:47:43.709 # Connection with slave 127.0.0.1:6380 lost.
22024:M 17 Oct 20:48:04.603 * Slave 127.0.0.1:6380 asks for synchronization
22024:M 17 Oct 20:48:04.603 * Partial resynchronization request from
127.0.0.1:6380 accepted. Sending 0 bytes of backlog starting from offset
565403937.
```

Slave:

```
21894:S 17 Oct 20:48:03.697 # Connection with master lost.
21894:S 17 Oct 20:48:03.697 * Caching the disconnected master state.
21894:S 17 Oct 20:48:04.602 * Connecting to MASTER 127.0.0.1:6379
21894:S 17 Oct 20:48:04.602 * MASTER <-> SLAVE sync started
21894:S 17 Oct 20:48:04.603 * Non blocking connect for SYNC fired the event.
21894:S 17 Oct 20:48:04.603 * Master replied to PING, replication can continue
...
21894:S 17 Oct 20:48:04.603 * Trying a partial resynchronization (request 8
f7b9821477006200651baef11d6af7451dede3d:565403937).
21894:S 17 Oct 20:48:04.603 * Successful partial resynchronization with master
.
21894:S 17 Oct 20:48:04.603 * MASTER <-> SLAVE sync: Master accepted a Partial
Resynchronization.
```

6. 为了准备第二个场景，我们关闭主从复制，清除从实例中的所有数据。然后，再次配置主从复制：

```
127.0.0.1:6380> SLAVEOF NO ONE
127.0.0.1:6380> FLUSHALL
127.0.0.1:6380> SLAVEOF 127.0.0.1 6379
```

7. 在主从复制的建立过程中，我们导入大量的数据：

```
for i in `seq 5`
do
nohup node generator.js hash 1000000 session &
done
```

8. 等待一段时间，然后再次检查主实例和从实例的日志：

```
Master:
16027:M 18 Oct 14:20:40.767 * Slave 127.0.0.1:6380 asks for synchronization
16027:M 18 Oct 14:20:40.767 * Partial resynchronization not accepted:
Replication ID mismatch (Slave asked for '40
f921547ba9b599dfff22d8d1fe2d7b03f284361', my replication IDs are '774
fce0f0ffe5afad272937699d714949cd9e8' and
'0000000000000000000000000000000000000000000000000000000000000000')
16027:M 18 Oct 14:20:40.767 * Starting BGSAVE for SYNC with target: disk
16027:M 18 Oct 14:20:40.826 * Background saving started by pid 22756
16027:M 18 Oct 14:21:04.240 # Client id=51 addr=127.0.0.1:41853 fd=7 name= age
=24 idle=24 flags=S db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-free=0 obl
=16384 oll=16362 omem=268435458 events=r cmd=psync scheduled to be closed
ASAP for overcoming of output buffer limits.
16027:M 18 Oct 14:21:04.256 # Connection with slave 127.0.0.1:6380 lost.
16027:M 18 Oct 14:22:04.896 * Slave 127.0.0.1:6380 asks for synchronization
16027:M 18 Oct 14:22:04.896 * Full resync requested by slave 127.0.0.1:6380
16027:M 18 Oct 14:22:04.896 * Can't attach the slave to the current BGSAVE.
Waiting for next BGSAVE for SYNC
22756:C 18 Oct 14:22:13.932 * DB saved on disk
22756:C 18 Oct 14:22:13.949 * RDB: 5091 MB of memory used by copy-on-write
16027:M 18 Oct 14:22:14.049 * Background saving terminated with success
16027:M 18 Oct 14:22:14.049 * Starting BGSAVE for SYNC with target: disk
16027:M 18 Oct 14:22:14.111 * Background saving started by pid 22845
22845:C 18 Oct 14:23:46.883 * DB saved on disk
22845:C 18 Oct 14:23:46.912 * RDB: 22 MB of memory used by copy-on-write
16027:M 18 Oct 14:23:47.031 * Background saving terminated with success
```

```

16027:M 18 Oct 14:24:40.644 * Synchronization with slave 127.0.0.1:6380
succeeded

Slave:
16022:S 18 Oct 14:20:40.765 * Connecting to MASTER 127.0.0.1:6379
16022:S 18 Oct 14:20:40.765 * MASTER <-> SLAVE sync started
16022:S 18 Oct 14:20:40.765 * Non blocking connect for SYNC fired the event.
16022:S 18 Oct 14:20:40.767 * Master replied to PING, replication can continue
...
16022:S 18 Oct 14:20:40.767 * Trying a partial resynchronization (request 40
f921547ba9b599dff22d8d1fe2d7b03f284361:607439042).
16022:S 18 Oct 14:20:40.826 * Full resync from master: 774
fce0f0ffe5afad272937699d714949cdfd9e8:3644738860
16022:S 18 Oct 14:20:40.827 * Discarding previously cached master state.
16022:S 18 Oct 14:22:04.896 # Timeout receiving bulk data from MASTER... If
the problem persists try to set the 'repl-timeout' parameter in redis.conf
to a larger value.
16022:S 18 Oct 14:22:04.896 * Connecting to MASTER 127.0.0.1:6379
16022:S 18 Oct 14:22:04.896 * MASTER <-> SLAVE sync started
16022:S 18 Oct 14:22:04.896 * Non blocking connect for SYNC fired the event.
16022:S 18 Oct 14:22:04.896 * Master replied to PING, replication can continue
...
16022:S 18 Oct 14:22:04.896 * Partial resynchronization not possible (no
cached master)
16022:S 18 Oct 14:22:14.111 * Full resync from master: 774
fce0f0ffe5afad272937699d714949cdfd9e8:5669698739
16022:S 18 Oct 14:23:47.031 * MASTER <-> SLAVE sync: receiving 6503393438
bytes from master
16022:S 18 Oct 14:24:40.656 * MASTER <-> SLAVE sync: Flushing old data
16022:S 18 Oct 14:24:40.656 * MASTER <-> SLAVE sync: Loading DB in memory
16022:S 18 Oct 14:26:08.477 * MASTER <-> SLAVE sync: Finished with success

```

5.4.3 工作原理

在配置 Redis 的复制机制的案例中，我们了解到，在处理完写入命令后，主实例会将这些命令发送给它的从实例以便实现主从实例之间的数据同步。从主实例的角度来看，要判断从实例是否仍在正常运行，就需要每隔一段固定的时间向从实例发送一个 PING 命令。我们可以通过配置文件或

redis-cli 修改 repl-ping-slave-period 参数来调整这个间隔。PING 命令的默认时间间隔是 10 秒。从从实例的角度来看，从实例每秒会向主实例发送 REPLCONF ACK {offset} 来报告它的复制偏移量。对于 PING 和 REPLCONF ACK 来说，都可以通过 repl-timeout 指定超时时间。复制的默认超时时间是 60 秒。如果两次 PING 或 REPLCONF ACK 之间的间隔时间比超时时间长，或者 repl-timeout 期间主实例和从实例之间没有数据流量，那么主从实例之间的复制连接将被断开。

第一个测试中，在 16:15:54 的时候，我们把主实例阻塞了 80 秒，比超时时间长。当超时时间到达后，从实例发现没有收到主实例的数据或 PING，因而在 16:16:52.823 时断开了与主实例的连接。之后，从实例直到 16:17:14.353 前一直试图重新建立与主实例的连接，随后主实例从挂起中唤醒并给了从实例一个回应。此后，主从复制得以继续。在挂起主实例之后，我们让从实例挂起了 80 秒。这时，可以得到类似的日志；但不同的部分是，当超时时间到达后，这次轮到了主实例切断复制连接。当从实例被唤醒后，发现与主实例的连接已经丢失了，所以它又重新尝试了一次连接到主实例。

第二个测试场景是另一个常见的生产问题。正如我们在配置 Redis 的复制机制的案例中所描述的那样，在主从复制建立时，主实例首先会以 RDB 文件的形式转储其内存并将其发送给从实例。当从实例接收完 RDB 文件时，它会将 RDB 文件加载到它的内存中。在这些步骤执行期间，对主实例的所有写入命令会被缓存在一个称为从客户端缓冲区 (Slave client buffer) 的特殊缓冲区中。在 RDB 加载后，主实例会将这个缓冲区的内容发送给从实例。

这个缓冲区的大小是有限制的。如果超出了限制，就会导致复制重新开始。从客户端缓冲区的默认大小限制为 slave 256 MB 64 MB 60。其中，“slave”表示这是从实例的缓冲区大小。值 256MB 是一个硬性的限制，一旦缓冲区大小达到这个限制，会立即关闭从实例的连接。64MB 和 60 作为一个整体则是一个软性的限制。其含义是，当缓冲区的大小超过 64MB 持续 60 秒时，主实例会关闭连接。

通过在数据同步过程中调用 CLIENT LIST 命令，我们可以捕获正在执行 sysrc 或 psysrc (cmd=sysrc/psysrc) 且而标志为 S (flag=S) 的客户端。然后，我们可以从 omem 指标中获得从实例缓冲区所使用的内存空间：

```
127.0.0.1:6379> CLIENT LIST
id=115 addr=127.0.0.1:46731 fd=11 name= age=41 idle=38 flags=S db=0 sub=0 psub
=0 multi=-1 qbuf=0 qbuf-free=0 obl=16382 oll=123 omem=2009753 events=r cmd
=psync
```

在之前的测试中，在同步过程启动之后，我们推送了一些超出了客户机缓冲区限制的大量测试数据。主实例检测到了这种情况并记录了 omem 指标，随后便很快地切断了连接：

```
16027:M 18 Oct 14:21:04.240 # Client id=51 addr=127.0.0.1:41853 fd=7 name= age
=24 idle=24 flags=S db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-free=0 obl
```

```
=16384 oll=16362 omem=268435458 events=r cmd=psync scheduled to be closed
ASAP for overcoming of output buffer limits.
16027:M 18 Oct 14:21:04.256 # Connection with slave 127.0.0.1:6380 lost.
```

60 秒（主从复制默认的超时时间）之后，从实例发现没有从主实例中获得数据，因此自动地重新启动了复制：

```
16022:S 18 Oct 14:22:04.896 # Timeout receiving bulk data from MASTER... If
the problem persists try to set the 'repl-timeout' parameter in redis.conf
to a larger value.
16022:S 18 Oct 14:22:04.896 * Connecting to MASTER 127.0.0.1:6379
```

5.4.4 更多细节

在实际的生产环境中，`repl-ping-slave-period` 的值必须小于 `repl-timeout` 的值。否则，只要每次主实例和从实例之间没有流量时就会造成复制超时。通常，因为 Redis 服务器的命令处理引擎是单线程的，所以阻塞操作可能会导致复制超时。为了防止复制超时发生，我们应该尽量避免使用会导致长时间阻塞的命令。在大多数情况下，`repl-timeout` 的默认值是足够的。

我们需要对第二种情况特别注意。主实例转储内存耗费的时间超过了 60 秒，但这并不会导致复制超时。同样，如果传输一个 RDB 文件的过程比复制超时时间还长，也不会导致复制超时。在这种情况下，复制并不会被中断。下面的示例显示了数据传输大约持续了 23 分钟，并最终复制成功：

```
16022:S 18 Oct 21:13:56.517 * MASTER <-> SLAVE sync: receiving 3983178656
bytes from master
16022:S 18 Oct 21:36:35.325 * MASTER <-> SLAVE sync: Flushing old data
16022:S 18 Oct 21:36:35.325 * MASTER <-> SLAVE sync: Loading DB in memory
16022:S 18 Oct 21:37:09.933 * MASTER <-> SLAVE sync: Finished with success
```

在本案例中，我们可以使用通过 root 用户执行 Linux 的 `tc` 命令设置本地回环线路（loopback）的延时来轻松地复现上述场景：

```
# tc qdisc add dev eth0 root netem delay 200ms
```

另外，在从实例中缓慢地加载 RDB 文件并不会阻塞从实例，也不会导致复制超时，如下例所示：

```
13760:S 16 Oct 23:12:52.414 * MASTER <-> SLAVE sync: receiving 5289155794
bytes from master
13760:S 16 Oct 23:13:37.637 * MASTER <-> SLAVE sync: Flushing old data
```

```
13760:S 16 Oct 23:13:37.637 * MASTER <-> SLAVE sync: Loading DB in memory
13760:S 16 Oct 23:23:20.231 * MASTER <-> SLAVE sync: Finished with success
```

总而言之，我们可以从 Redis 的源代码中了解到，只有在执行重新同步时主从实例之间没有数据传输，或者在超时时间内主/从实例无法接收 PING/REPLCONF ACK 时，复制超时才会被触发。

对于客户端缓冲区的大小，在高写入流量的生产环境中，我们可以通过调用 CONFIG SET 命令或修改配置文件并重新启动实例的方法来将其设置得更大。当我们使用 CONFIG SET 命令设置此值时，必须非常谨慎。诸如 MB 或 GB 等单位是不支持的。下面的示例把缓冲区大小的硬性限制和弹性限制分别设为 512MB 和 128MB/120 秒：

```
127.0.0.1:6379> config set client-output-buffer-limit 'slave 536870912
134217728 120'
```

最后值得一提的是，Redis 的键过期机制是由主实例驱动的。也就是说，当主实例中的键到期时，它会向所有的从实例发送一个 DEL 命令。在同步期间，DEL 命令也会被放到复制客户端缓冲区中。

此外，如果我们要把从实例设置为可写的，那么应该非常谨慎。这是因为，在 Redis 4.0 之前，Redis 可写从实例上设置了超时时间的键永远不会过期。这个问题已经在 Redis 4.0 RC3 及以后版本中得到了修复。

5.4.5 相关内容

- 我们还可以在以下链接中找到本案例中所提到配置的含义：<http://download.redis.io/redis-stable/-redis.conf>。

第 6 章

持久化

在本章中，我们将学习下列案例：

- 使用 RDB。
- 探究 RDB 文件。
- 使用 AOF。
- 探究 AOF 文件。
- RDB 和 AOF 的结合使用。

6.1 本章概要

由于 Redis 是在内存中存储数据的，所以当服务器重新启动时，所有的数据都将丢失。在上一章中，我们学习了如何配置 Redis 的主从同步实现由主实例中向从实例复制数据。对于数据冗余来说，Redis 的复制机制也可被用作备份数据的一种方式。

为了保证数据的安全，Redis 还提供了将数据持久化到磁盘中的机制。Redis 共有两种数据持久化类型：RDB 和 AOF。RDB 可以看作是 Redis 在某一个时间点上的快照（snapshot），非常适合于备份和灾难恢复。AOF 则是一个写入操作的日志，将在服务器启动时被重放。

在本章中，我们将介绍如何在 Redis 中启用和配置 RDB 及 AOF。我们还将学习 RDB 和 AOF 的文件格式，以及在 Redis 中配置持久化时与调优相关的重要配置参数。在本章的最后，我们将讨论如何在 Redis 中将 RDB 和 AOF 结合起来使用。

6.2 使用 RDB

正如我们在前面的案例中所提到的，对于一个像 Redis 这样的内存数据存储，启用持久化功能是防止数据丢失的最好方法。实现持久化的一种显而易见的方法是不停地对存储数据的内存进行快照，而这基本上正是 Redis 中 RDB 的工作方式。

在本案例中，我们将学习如何配置 RDB 持久化功能，并了解有关快照执行过程的更多细节。

6.2.1 准备工作

我们需要按照第 1 章 开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。

出于演示的目的，我们使用 FLUSHALL 命令清除所有的数据，然后利用第 5 章复制中复制机制的故障诊断的案例中介绍的 redis-random-data-generator 工具将一些测试数据导入到 Redis 实例中：

```
for i in `seq 10`
do
nohup node generator.js hash 1000000 session &
done
```

6.2.2 操作步骤

使用 RDB 的步骤如下：

1. 调用 CONFIG SET 命令，在一个正在运行的 Redis 实例上启用 RDB 持久化：

```
127.0.0.1:6379> CONFIG SET save "900 1"
OK
127.0.0.1:6379> CONFIG SET save "900 1 300 10 60 10000"
OK
```

2. 如果想永久性地启用 RDB 持久化，可以在 Redis 配置文件中设置 save 参数：

```
$ cat conf/redis.conf |grep "^save"
save 900 1
save 300 10
save 60 10000
```

3. 要在一个正在运行的 Redis 实例上禁用 RDB 持久化，可以使用 `redis-cli` 把 `save` 参数设为空字符串：

```
$ bin/redis-cli config set save ""  
OK
```

4. 要禁用 RDB 持久化，只需在配置文件中注释掉或删除 `save` 参数：

```
#save 900 1  
#save 300 10  
#save 60 10000
```

5. 通过 `redis-cli` 获取 `save` 选项的值可以判断是否启用了 RDB 功能。如果 `save` 选项是一个空字符串，则表示 RDB 功能是被禁用的。否则，则返回 RDB 快照的触发策略：

```
$ bin/redis-cli CONFIG GET save  
1) "save"  
2) "900 1 300 10 60 10000"
```

6. 我们也可以在 Redis 的数据目录中检查是否生成了 RDB 文件：

```
$ ls -l dump.rdb  
-rw-rw-r-- 1 redis redis 286 Oct 23 22:11 dump.rdb
```

7. 如果要手动执行 RDB 快照，在 `redis-cli` 中调用 `SAVE` 命令即可。`redis-cli` 将会被阻塞一段时间，然后命令提示符会返回如下内容：

```
127.0.0.1:6379> SAVE  
OK  
(23.25s)  
127.0.0.1:6379>
```

8. 或者，我们也可以调用 `BGSAVE` 命令来执行非阻塞的 RDB 转储：

```
127.0.0.1:6379> BGSAVE  
Background saving started
```

9. 我们可以使用 `ps -ef |grep redis` 命令检查转储进程的 pid：

```
$ps -ef |grep redis  
redis    10708 21158 91 21:21 ?        00:00:06 redis-rdb-bgsave 0.0.0.0:6379  
redis    21158      1  5 17:35 ?        00:12:46 bin/redis-server 0.0.0.0:6379
```


10. 在 RDB 转储过程中快速地导入一些测试数据:

```
$ node generator.js hash 100000
```

11. 检查 Redis 服务器的日志:

```
$ cat log/redis.log
21158:M 21 Oct 21:21:41.408 * Background saving started by pid 777
777:C 21 Oct 21:22:04.316 * DB saved on disk
777:C 21 Oct 21:22:04.346 * RDB: 322 MB of memory used by copy-on-write
21158:M 21 Oct 21:22:04.449 * Background saving terminated with success
```

12. 使用 `redis-cli` 执行 `INFO PERSISTENCE` 命令可以获取与持久化相关的指标和状态:

```
127.0.0.1:6379> INFO PERSISTENCE
# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:1
rdb_last_save_time:1508590312
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:21
rdb_current_bgsave_time_sec:20
rdb_last_cow_size:2637824
```

6.2.3 工作原理

RDB 就像是一台专门给 Redis 数据存储拍照的照相机。当满足触发策略时, Redis 会通过将所有数据转储到本地磁盘上一个文件中的方式给 Redis 中的数据拍一张“照片”。在详细介绍上述过程前,我们先来解释一下 `save` 参数的含义。`save` 参数决定了上一节中提到的 RDB 触发策略,这个值的格式是 $x_1, y_1, x_2, y_2, \dots$, 其含义是, 如果超过 y 个键发生改变且此时没有转储正在发生, 则在 x 秒后进行数据转储。

在前面的示例中, `900 1` 表示如果在 900 秒内有超过 1 个键发生了改变, 则会进行一次 RDB 快照。在 `save` 选项中, 可以同时使用多个策略来控制 RDB 快照执行的频率。

通过 `SAVE` 或 `BGSAVE` 命令可以手动启动一次 RDB 转储。这两个命令的不同之处在于, 前者使用 Redis 的主线程进行同步转储, 而后者则在后台进行转储。因为 `SAVE` 命令会阻塞 Redis 服务器, 所以永远不要在生产环境中使用。与之不同, 对于 `BGSAVE` 命令而言, Redis 的主线程将继续处理收到的命令, 同时会通过 `fork()` 系统调用创建一个子进程将转储数据保存到一个名为 `temp-<bgsave-`

`pid>.rdb` 的临时文件中。当转储过程结束后，这个临时文件会被重命名为由参数 `dbfilename` 定义的名字并覆盖由参数 `dir` 指定的本地目录中的旧转储文件。上述两个参数都可以通过 `redis-cli` 进行动态修改。

在使用 `ps` 命令列出的进程中，我们会发现名为 `redis-rdb-bgsave` 的子进程，该子进程就是 Redis 服务器创建出来用于执行 `BGSAVE` 命令的子进程。这个子进程会保存 `BGSAVE` 命令被处理时间点的所有数据。由于使用了写时复制（**Copy-On-Write, COW**）机制，所以子进程不需要使用 Redis 服务器所占用的同等数量的内存。

在转储过程中，我们可以使用 `INFO PERSISTENCE` 来获取当前正在进行的持久化过程的相关指标。在本例中，`rdb_bgsave_in_progress` 的值为 1，表示转储正在进行。`rdb_current_bgsave_time_sec` 的值表示当前正在进行的 `BGSAVE` 命令所持续的时间。

在子进程替换完旧转储文件后，它会根据子进程使用的私有数据量输出一条日志：

```
777:C 21 Oct 21:22:04.346 * RDB: 322 MB of memory used by copy-on-write
```

这个数值基本上指的是父进程与子进程相比执行了多少修改（包括读取缓冲区、写入缓冲区、数据修改等）。在示例中，我们在转储过程中导入了一些测试数据，因此写时复制使用的内存（在 `INFO` 命令中）并不是很小（对于一个有 2GB 数据的 Redis 实例，如果转储过程中没有数据写入，那么写时复制通常会使用 10-20MB 的内存）。这个指标还保存在 `INFO` 命令获得的 `rdb_last_cow_size` 中。

在 `BGSAVE` 命令完成后，最后一次 `BGSAVE` 的状态保存在指标 `rdb_last_bgsave_status` 中，而最后一次成功 `BGSAVE` 的 Unix 时间戳保存在 `rdb_last_save_time` 中：

```
127.0.0.1:6379> INFO PERSISTENCE
# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1508639640
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:20
rdb_current_bgsave_time_sec:-1
rdb_last_cow_size:15036416
```

6.2.4 更多细节

下面是一些与 RDB 持久化相关的配置选项。我们建议读者使用默认值，也就是将以下这些选项设置为 `yes`：

- `stop-writes-on-bgsave-error yes`：如果该选项设置为 `yes` 的话，那么作为保护机制，当 `BGSAVE` 失败时 Redis 服务器将停止接受写入操作。
- `rdbcompression yes`：压缩可以显著减小转储文件的大小，但会在 LZF 压缩时消耗更多的 CPU。
- `rdbchecksum yes`：在快照文件的末尾创建一个 CRC64 校验和。使用该选项将在保存和加载快照文件时额外消耗约 10% 的性能。将该选项设置为 `no` 可以获得最大性能，但同时也会降低对数据损坏的抵抗力。

由 `SAVE/BGSAVE` 命令生成的转储可以用作数据备份文件。我们可以使用操作系统中的 `crontab` 定期将 RDB 文件复制到本地目录或诸如 Amazon S3/HDFS 的远程分布式文件系统中，供日后恢复使用：

```
cp /redis/data/dump.rdb /somewhere/safe/dump.$(date +%Y%m%d%H%M).rdb
```

如果要还原 RDB 快照，我们需要把快照文件复制到 `dir` 选项指定的位置，并且保证启动 Redis 实例的用户有该文件的读/写权限。之后，我们需要通过 `SHUTDOWN NOSAVE` 命令停止实例，并将新转储文件重命名为 `dbfilename` 选项定义的名字。重新启动后，数据就会从备份文件中加载并还原回 Redis 了。

6.2.5 相关内容

- 更多有关 RDB 持久化的讨论，请参阅：<https://redis.io/topics/persistence>。
- 读者可能对写时复制在操作系统中的工作方式感兴趣；更多细节请参阅：<https://en.wikipedia.org/wiki/Copy-on-write>。

6.3 探究 RDB 文件

我们已经知道到，Redis 通过持久化将其内存中的数据转储到一个默认名为 `dump.rdb` 的文件中。对于初学者来说，理解上一个案例中提到的关键内容已经足够了。对于还想更深入地了解 RDB 文件是如何以二进制的形式表示 Redis 中内存数据的极客来说，学习 RDB 文件的格式是大有裨益的。在本案例中，我们将深入学习 RDB 文件的格式，并了解在学习了 RDB 格式后还能实现什么。

6.3.1 准备工作

我们需要按照第 1 章 开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。

我们需要使用二进制编辑器来查看二进制文件的格式。在 Ubuntu Linux 操作系统中，bvi 是最常用的二进制编辑器。我们可以使用如下的方法进行安装：

```
$ sudo apt-get install bvi
```

对于 macOS 操作系统，我们可以从<https://github.com/ridiculousfish/HexFiend/releases>下载 HexFiend。

出于演示的目的，我们需要使用 FLUSHALL 命令清除所有数据。

我们将以十六进制来浏览 RDB 文件，所以需要在 ASCII 表中查找 ASCII 字符。RapidTables 网站（<http://www.rapidtables.com/code/text/ascii-table.htm>）是一个很好的辅助工具。

6.3.2 操作步骤

接下来，让我们按照以下步骤学习 RDB 文件的格式：

1. 设置一个字符串类型的键值对：

```
127.0.0.1:6379> SET key value
```

```
OK
```

2. 设置一个列表类型的键值对：

```
127.0.0.1:6379> LPUSH listkey v1 v2
```

```
(integer) 2
```

3. 设置一个哈希类型的键值对：

```
127.0.0.1:6379> HSET hashkey k1 v1 k2 v2
```

```
(integer) 2
```

4. 设置一个集合类型的键值对：

```
127.0.0.1:6379> SADD setkey v1 v2
```

```
(integer) 2
```

5. 设置一个有序集合类型的键值对：

```
127.0.0.1:6379> ZADD zset 1 v1 2 v2
```

```
(integer) 2
```

6. 在 redis-cli 中调用 SAVE 命令触发 RDB 转储：

```
127.0.0.1:6379> SAVE
```

```
OK
```

7. 使用 bvi 编辑器打开刚刚生成的 dump.rdb 文件：

```
$ bvi dump.rdb
```

dump.rdb 的内容如图 6.1 所示。

```
00000000 57 45 44 49 53 30 30 30 38 FA 09 72 65 64 69 73 REDIS0008..redis
00000010 2D 76 65 72 05 34 2E 30 2E 31 FA 0A 72 65 64 69 -ver.4.0.1..redi
00000020 73 2D 62 69 74 73 C0 40 FA 05 63 74 69 6D 65 C2 s-bits.@.ctime.
00000030 7B F8 ED 59 FA 08 75 73 65 64 2D 6D 65 6D C2 98 [.Y..used-mem..
00000040 A1 0C 00 FA 0C 61 6F 66 2D 70 72 65 61 6D 62 6C ....aof-preambl
00000050 65 C0 00 FA 07 72 65 70 6C 2D 69 64 28 65 30 61 e....repl-id(e0a
00000060 35 39 33 62 37 33 37 36 31 37 63 61 36 63 61 63 593b737617ca6cac
00000070 31 30 64 38 63 64 35 64 62 33 64 34 30 38 64 35 10d8cd5db3d408d5
00000080 61 30 30 33 65 FA 0B 72 65 70 6C 2D 6F 66 66 73 a003e..repl-offs
00000090 65 74 C1 CF 50 FE 00 FB 05 00 0D 07 68 61 73 68 et..P.....hash
000000A0 6B 65 79 1B 1B 00 00 16 00 00 04 00 00 02 key.....
000000B0 6B 31 04 02 76 31 04 02 68 32 04 02 76 32 FF 0E k1..v1..k2..v2..
000000C0 07 6C 69 73 74 6B 65 79 01 13 13 00 00 00 0E 00 ..listkey.....
000000D0 00 00 02 00 00 02 76 32 04 02 76 31 FF 02 06 73 .....v2..v1...s
000000E0 65 74 6B 65 79 02 02 76 31 02 76 32 00 03 6B 65 etkey..v1.v2..ke
000000F0 79 05 76 61 6C 75 65 0C 04 7A 73 65 74 17 17 00 y.value..zset...
00000100 00 00 14 00 00 00 04 00 00 02 76 31 04 F2 02 02 .....v1....
00000110 76 32 04 F3 FF FF 4E CB 90 01 E1 BA 21 35 v2....N.....!5
```

图 6.1 dump.rdb 的内容

6.3.3 工作原理

在前面的例子中，我们把一些简单的测试数据导入了 Redis 实例中。之后，通过 SAVE 命令创建了一个 RDB 转储文件。

接下来，让我们使用 Linux 中的二进制编辑器 bvi 进一步学习 RDB 文件究竟包含了什么内容。RDB 文件的所有内容都已经被 bvi 转换成了十六进制格式。通过在 RapidTables (<https://www.rapidtables.com/code/text/ascii-table.html>) 的 ASCII 表中查找前五个十六进制数字 (52 45 44 49 53)，我们会发现前五个字符是“REDIS”：

ASCII Quick Lookup

Enter Character / Decimal code / Hex code / Binary code to get other values:

Char:	HTML Code:	Escape Code:
R	R	\x52
Decimal:	Hex:	Binary:
82	52	01010010
✖ Reset		

‘REDIS’ 字符串后面的四个字符是 30 30 30 38，表示 RDB 格式的版本是 0008。REDIS0008 是 RDB 文件的魔数字符串。

在魔数字符串之后，保存了 RDB 文件中 8 种元数据：

- redis-ver: Redis 实例的版本。
- redis-bits: 运行 Redis 实例的主机的架构。该元数据的有效值是 64 或 32。
- ctime: RDB 创建时的 Unix 时间戳。
- used-mem: 转储时使用的内存大小。
- repl_stream_db: 复制链中数据库的索引，只在启用了复制机制时才存在。
- aof-preamble: 是否在 AOF 文件的开头放置 RDB 快照（译者注：即开启混合持久化）。
- repl-id: 主实例的 ID（replication ID）。
- repl-offset: 主实例的偏移（replication offset）。

另外，RDB 文件中的 FA 是一个辅助操作代码。在辅助代码之后总会跟一个键值对。

在保存了 Redis 服务器中每个数据库的元数据后，转储进程还将保存数据库索引（本例中为 FE 00，表示数据库的索引是 0）、哈希大小调整代码、数据库大小调整操作代码，以及过期大小调整操作代码。随后，通过迭代将 Redis 中的每个元素写入转储文件。Redis 中一个键值对的格式如下：

Expiration	Timestamp	Data Type	Key length	Key	Value Length	Value
------------	-----------	-----------	------------	-----	--------------	-------

由于本书篇幅的限制，我们不会详细介绍上述的数据格式。如果读者对此感兴趣，可以参考源文件 rdb.c 和 rdb.h。

RDB 文件以 EOF 和 CRC64 校验和结束。

6.3.4 更多细节

了解了 RDB 格式后，我们就可以创建自己的 Redis 工具来进行内存分析、数据导出/导入，以及合并多个 Redis 实例等工作了。如果在 GitHub 上搜索 ‘Redis RDB’，将会了解到更多关于 RDB 格式的信息。

6.3.5 相关内容

- 请参阅如下链接了解关于 RDB 格式的更多信息：<https://github.com/sripathikrishnan/redis-rdb-tools/wiki/Redis-RDB-Dump-File-Format>和<https://github.com/leonchen83/redis-replicator/wiki/RDB-dump-data-format>。
- Redis 的作者 Antirez 也介绍了 RDB 的设计，请参阅：<https://redis.io/topics/rdd-2>。

6.4 使用 AOF

在前两个案例中，我们学习了 Redis 数据持久化机制中的 RDB 方式。然而，使用 RDB 进行数据持久化并不能提供非常强的一致性。在上一案例中曾提到，一个 RDB 数据转储文件仅包含某个时间点上的 Redis 数据快照。虽然我们可以定期地将数据转储到 RDB 文件中，但当 Redis 进程崩溃或者出现硬件故障时，两次 RDB 转储之间的数据将会永久丢失。AOF（append-only file）是一种只记录 Redis 写入命令的追加式日志文件。因为每个写入命令都会被追加到文件中，所以 AOF 的数据一致性比 RDB 更高。在本案例中，我们将展示如何在 Redis 中启用 AOF，并介绍一些 AOF 的重要配置参数。

6.4.1 准备工作

我们需要按照第 1 章 开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。

6.4.2 操作步骤

使用 AOF 的步骤如下：

1. 调用 CONFIG SET 命令，在一个正在运行的 Redis 实例上启用 AOF 持久化：

```
127.0.0.1:6379> CONFIG SET appendonly yes
OK
```

2. 如果想永久性地启用 AOF 持久化，可以在 Redis 配置文件中添加 appendonly yes，然后重新启动服务器：

```
$ cat conf/redis.conf |grep "^appendonly"
appendonly yes
```

3. 要在一个正在运行的 Redis 实例上禁用 AOF 持久化，可以把 appendonly 参数设为 no：

```
$ bin/redis-cli config set appendonly no
OK
```

4. 要永久地禁用 AOF 持久化，只需在配置文件中设置 appendonly no 并重新启动服务器：

```
$ cat conf/redis.conf |grep "^appendonly"
appendonly no
```

5. 使用 `INFO PERSISTENCE` 命令并检查 AOF 相关的内容可以判断是否启用了 AOF 功能：

```
127.0.0.1:6379> INFO PERSISTENCE
# Persistence
loading:0
...
aof_enabled:1
aof_rewrite_in_progress:0
```

6. 或者也可以检查在 Redis 的数据目录中是否生成了 AOF 文件：

```
$ ls -l
-rw-r--r-- 1 root root 233 Oct 22 22:16 appendonly.aof
```

6.4.3 工作原理

当 AOF 功能被启用时，Redis 将会在数据目录中创建 AOF 文件。AOF 文件的默认文件名是 `appendonly.aof`，并可以通过配置文件中的 `appendfilename` 参数进行修改。同时，Redis 还会将当前内存中的数据填充到这个 AOF 文件中。

每当 Redis 服务器接收到一个会实际修改内存数据的写入命令时，Redis 会将该命令追加到 AOF 文件中。但是，如果我们深入研究 AOF 文件的写入过程，就会发现操作系统实际上维护了一个缓冲区，Redis 的命令首先会被写入到这个缓冲区中。而缓冲区中的数据必须被刷新到磁盘中才能被永久保存。这个过程是通过 Linux 系统调用 `fsync()` 完成的，这是一个阻塞调用，只有磁盘设备报告缓冲区中的数据写入完成后才会返回。

在将命令追加到 AOF 文件时，我们可以通过 Redis 的配置参数 `appendfsync` 来调整调用 `fsync()` 的频率。该参数共有三个选项：

- `always`：对每个写入命令都调用 `fsync()`。这个选项可以确保在发生意外的服务器崩溃或硬件故障时，只丢失一个命令。但是，由于 `fsync()` 是一个阻塞调用，Redis 的性能会受到物理磁盘写入性能的限制。将 `appendfsync` 设置为 `always` 是不明智的，因为 Redis 服务器的性能会显著降低。
- `everysec`：每秒调用一次 `fsync()`。采用这一选项时，在意外灾难事件中只有一秒钟的数据会丢失。我们建议读者使用此选项以在数据鲁棒性和性能之间进行平衡。
- `no`：永远不调用 `fsync()`。采用该选项时，将由操作系统决定何时将数据从缓冲区写入到磁盘。在大多数 Linux 系统中，这个频率是每 30 秒。

当 Redis 服务器关闭时，`fsync()` 会被显式地调用，以确保写入缓冲区中的所有数据都会被刷新到磁盘。

当 Redis 服务器启动时，AOF 文件会被用来恢复数据。Redis 只需要通过读取命令来重放文件，并将它们逐个应用到内存中即可。待所有命令都被处理完之后，数据也就被重建好了。

6.4.4 更多细节

随着 Redis 将写入命令追加到 AOF 文件中，AOF 文件的大小可能会显著增加；而这会拖慢 Redis 启动时的数据重建过程。Redis 提供了一种机制，即通过 AOF 重写（AOF rewrite）来压缩 AOF 文件。读者可能已经猜到，Redis 中的某些键可能已经被删除或过期了，所以可以在 AOF 文件中将其清除；同时，某些键的值可能被更新了多次，但只有最新的值才需要存储在 AOF 文件中。这就是 AOF 重写中数据压缩的基本思想。我们可以使用 BGREWRITEAOF 命令来启动重写过程，或者让 Redis 自动执行 AOF 重写。我们将在后续案例中讨论 AOF 重写。

如果操作系统崩溃，那么 AOF 文件的最后可能会损坏或截断。Redis 提供了一个工具，redis-check-aof，用于修复损坏的文件。如果需要修复一个 AOF 文件，只需运行：

```
$ bin/redis-check-aof --fix appendonly.aof
```

6.4.5 相关内容

- 关于 AOF 持久化的更多讨论，请参阅：<https://redis.io/topics/persistence>。
- Redis 的作者撰写的一篇博文解释了数据的持久化，请参阅：<http://oldblog.antirez.com/post/redis-persistence-demystified.html>。
- 关于 fsync()，请参阅：<https://linux.die.net/man/2/fsync>。

6.5 探究 AOF 文件。

在之前的案例中，我们学习了如何在 Redis 中配置 AOF 持久化。在本案例中，我们将学习 AOF 文件的内容并解释 AOF 重写的过程。

6.5.1 准备工作

我们需要按照第 1 章 开始使用 Redis 中启动和停止 Redis 的案例中描述的步骤安装一个 Redis 服务器。

出于演示的目的，我们首先需要禁用 AOF，并使用 FLUSHALL 命令清除所有数据。

6.5.2 操作步骤

接下来，让我们按照以下的步骤来学习 AOF 文件的格式。

1. 启用 AOF 持久化：

```
127.0.0.1:6379> CONFIG SET appendonly yes
OK
```

2. 由于 Redis 中没有数据，我们应该会得到一个空的 AOF 文件：

```
$ ls -l
-rw-rw---- 1 redis redis 0 Oct 23 20:47 appendonly.aof
```

3. 在 redis-cli 中运行以下命令：

```
127.0.0.1:6379> SET k1 v1
OK
127.0.0.1:6379> INCR counter
(integer) 1
127.0.0.1:6379> INCR counter
(integer) 2
127.0.0.1:6379> SET k2 v2
OK
127.0.0.1:6379> DEL k1
(integer) 1
127.0.0.1:6379> DEL k3
(integer) 0
127.0.0.1:6379> HSET mykey f1 v1 f2 v2
(integer) 2
```

4. 使用文本编辑器打开 AOF 文件：

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n
*3\r\n$3\r\nset\r\n$2\r\nk1\r\n$2\r\nv1\r\n
*2\r\n$4\r\nincr\r\n$7\r\ncounter\r\n
*2\r\n$4\r\nincr\r\n$7\r\ncounter\r\n
...
```

5. 在 redis-cli 中执行 BGREWRITEAOF 命令：

```
127.0.0.1:6379> BGREWRITEAOF
```

Background append only file rewriting started

6. 再次使用文本编辑器查看 AOF 文件：

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n
*3\r\n$3\r\nSET\r\n$7\r\ncounter\r\n$1\r\n2\r\n
*3\r\n$3\r\nSET\r\n$2\r\nk2\r\n$2\r\nv2\r\n
*6\r\n$5\r\nHMSET\r\n$5\r\nmykey\r\n$2\r\nf1\r\n$2\r\nv1\r\n$2\r\nf2\r\n$2\r\n
nv2\r\n
```

6.5.3 工作原理

在准备工作一节中，我们禁用了 AOF 持久化并清除了 Redis 中的所有数据。在那之后，重新启用 AOF 持久化时将创建一个空的 AOF 文件。请注意，在清除数据前，我们必须先禁用 AOF 持久化，否则 AOF 文件不会是空的（除非执行 AOF 重写）。

在第 3 步中，我们执行了七个命令来操作 Redis 中的键：

- SET k1 v1
- INCR counter
- INCR counter
- SET k2 v2
- DEL k1
- DEL k3
- HSET mykey f1 v1 f2 v2

当打开 AOF 文件进行查看时，我们发现文件的内容看上去就是我们刚执行的命令，其格式为第 1 章开始使用 Redis 中理解 Redis 通信协议的案例中所介绍的 RESP 格式。

但其中有两处例外：

- 最开始处的命令 SELECT 0，表示选择索引为 0 的数据库。Redis 支持多个数据库，这些数据库由 0 到 15 的数字指定，默认的数据库索引是 0。
- 命令 DEL k3 不在 AOF 文件中，因为这个命令不会使数据库中的任何数据发生变化（键 k3 是不存在的）。

在第 5 步中，我们执行了一次 AOF 重写。在重写完成后，我们可以看到 AOF 文件被压缩并发生了如下的变化：

- 两个 INCR 命令被一个 SET 命令替代。

- 操作键 `k1` 的命令消失，因为 `k1` 已经被删除了。Redis 通过一个算法来压缩和重写 AOF 文件，其他的例子还包括：多个 `SET` 命令被一个 `MSET` 命令替换、操作已过期键的命令被移除，等等。

Redis 主进程会创建一个子进程来执行 AOF 重写。这个子进程会创建一个新的 AOF 文件来存储重写的结果，以防止重写操作失败时影响旧的 AOF 文件。父进程则继续响应请求并将命令追加到旧的 AOF 文件中。在创建子进程时，由于使用了写时复制机制，所以子进程不会占用与父进程相同数量的内存。但是，由于写时复制机制的限制，子进程不能访问在其被创建后新产生的数据。Redis 通过如下方式来解决这个问题：在子进程创建后，主进程（父进程）将接收到的命令写入一个名为 `aof_rewrite_buf_blocks` 的缓冲区。当子进程完成新 AOF 文件的重写后，它会向父进程发送一个信号。父进程会把 `aof_rewrite_buf_blocks` 缓冲区中的命令写入到新 AOF 文件中，然后用新文件替换旧的 AOF 文件。

6.5.4 更多细节

除了用 `BGREWRITEAOF` 命令手动触发 AOF 重写之外，我们还可以配置 Redis 自动地执行 AOF 重写。这主要涉及以下两个配置参数：

- `auto-aof-rewrite-min-size`：如果文件大小小于此值则 AOF 重写不会被触发。默认值是 64MB。
- `auto-aof-rewrite-percentage`：Redis 将记录最后一次 AOF 重写操作的文件大小。如果当前的 AOF 文件大小增长超过了这个百分比，则触发一次重写。将此值设置为 0 将禁用自动 AOF 重写（*Automatic AOF Rewrite*）。默认值是 100。

`INFO PERSISTENCE` 命令提供了有关 AOF 重写的很多信息。例如，`aof_last_bgrewrite_status` 表示最后一次 AOF 重写操作的状态，`aof_base_size` 则是最后一个 AOF 文件的大小，`aof_rewrite_buffer_length` 则是前面提到的 `aof_rewrite_buf_blocks` 的大小：

```
127.0.0.1:6379> INFO PERSISTENCE
...
aof_enabled:1
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:0
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok
aof_last_write_status:ok
aof_last_cow_size:6393856
aof_current_size:143
```



```
aof_base_size:143
aof_pending_rewrite:0
aof_buffer_length:0
aof_rewrite_buffer_length:0
aof_pending_bio_fsync:0
aof_delayed_fsync:0
```

6.5.5 相关内容

- 对其他 AOF 持久化相关信息的解释，请参阅：<https://redis.io/commands/info>。

6.6 RDB 和 AOF 的结合使用

在本章的前几个案例中，我们学习了 RDB 和 AOF 持久化。在涉及数据持久化时，我们总是需要考虑如下几个因素：意外停机时的数据丢失、保存数据时的性能开销、持久化文件的大小以及数据恢复的速度。对于 RDB 而言，两次快照之间写入的数据可能会丢失。当写入流量较大且数据集也很大时，RDB 中系统调用 `fork()` 的延迟和内存开销可能会变成一个问题。不过，与 AOF 相比，RDB 转储文件占用的磁盘空间更少，并且从 RDB 转储中恢复数据的速度更快。事实上，我们可以同时启用这两个功能。

在本案例中，我们将学习如何充分利用这两种持久化方法的优点。

6.6.1 准备工作

我们需要按照第 1 章 开始使用 *Redis* 中启动和停止 *Redis* 的案例中描述的步骤安装一个 *Redis* 服务器。此外，我们还需要对 RDB 和 AOF 有基本的了解。

6.6.2 操作步骤

1. 在 *Redis* 的配置文件中进行如下的配置可以同时启用 RDB 和 AOF 持久化功能：

```
$ cat conf/redis.conf | egrep "^save|^append"
save 900 1
save 300 10
save 60 10000
appendonly yes
appendfilename "appendonly.aof"
appendfsync everysec
```

2. 使用 `DEBUG POPULATE` 命令将一些测试数据导入到 Redis 服务器中，并同时执行 RDB 和 AOF 的数据持久化操作。在这些操作之后，设置一个简单的字符串类型的键值对：

```
127.0.0.1:6379> DEBUG POPULATE 1000000
OK
(1.61s)
127.0.0.1:6379> SAVE
OK
127.0.0.1:6379> BGREWRITEAOF
Background append only file rewriting started

127.0.0.1:6379> SET FOO BAR
OK
```

3. 查看 RDB 和 AOF 文件：

```
$ ls -l dump.rdb appendonly.aof
-rw-rw-r-- 1 redis redis 48676857 Oct 25 10:25 appendonly.aof
-rw-rw-r-- 1 redis redis 24777946 Oct 25 10:24 dump.rdb
```

4. 查看 RDB 和 AOF 文件的大小：

```
$ du -sm dump.rdb appendonly.aof
24    dump.rdb
47    appendonly.aof
```

5. 通过 Vim 编辑器查看 AOF 文件的前 20 行：

```

1 *2
2 $6
3 SELECT
4 $1
5 0
6 *3
7 $3
8 SET
9 $10
10 key:199713
11 $12
12 value:199713
13 *3
14 $3
15 SET
16 $10
17 key:547158
18 $12
19 value:547158
20 *3
21 $3
22 SET

```

然后，通过 Vim 编辑器查看 AOF 文件的最后 10 行：

```

7000001 SET
7000002 $10
7000003 key:562743
7000004 $12
7000005 value:562743
7000006 *2
7000007 $6
7000008 SELECT
7000009 $1
7000010 0
7000011 *3
7000012 $3
7000013 SET
7000014 $3
7000015 FOO
7000016 $3
7000017 BAR

```

6. 将配置参数 `aof-use-rdb-preamble` 改为 `yes`，然后再次使用 `BGREWRITEAOF` 命令触发 AOF 文件重写：

```
127.0.0.1:6379> CONFIG SET aof-use-rdb-preamble yes
```

```
OK
```



```
127.0.0.1:6379> BGREWRITEAOF
```

```
Background append only file rewriting started
```

7. 查看 RDB 和 AOF 文件的大小：

```
$ du -sm dump.rdb appendonly.aof
```

```
24      dump.rdb
```

```
24      appendonly.aof
```

8. 通过 Vim 编辑器查看 AOF 文件的前 20 行：

```
1 [EDIS0008 redis-ver^E4.0.1[33m 2 redis-bits[34m^Ectime±
[34m^Hused-mem[34m^D4m^Laof-preamble4m^A4m^A
Grep1-id(e5da1042134a3e4fe84be1db46a6ea4849c5c5824m^Krep1-offs
t4m^4m^4m~@^@^AOB@^@^@
3 key:455483^Lvalue:455483^@
4 key:570281^Lvalue:570281^@
5 key:356462^Lvalue:356462^@ key:67227^Kvalue:67227^@
6 key:147364^Lvalue:147364^@
7 key:255808^Lvalue:255808^@
8 key:252551^Lvalue:252551^@
9 key:591381^Lvalue:591381^@
10 key:748670^Lvalue:748670^@
11 key:213878^Lvalue:213878^@
12 key:834908^Lvalue:834908^@
13 key:421798^Lvalue:421798^@
14 key:697478^Lvalue:697478^@
15 key:107675^Lvalue:107675^@ key:94876^Kvalue:94876^@
16 key:779533^Lvalue:779533^@
17 key:538367^Lvalue:538367^@
18 key:561941^Lvalue:561941^@
19 key:163971^Lvalue:163971^@
20 key:501156^Lvalue:501156^@
```

然后，通过 Vim 编辑器查看 AOF 文件的最后 10 行：

```
908999 key:412162^Lvalue:412162^@
909000 key:926399^Lvalue:926399^@
909001 key:902673^Lvalue:902673^@
909002 key:404168^Lvalue:40416834m^Q[34m^_AH*2^M
909003 $6^M
909004 SELECT^M
909005 $1^M
909006 0^M
909007 *3^M
909008 $3^M
909009 set^M
909010 $3^M
909011 foo^M
909012 $3^M
909013 bar^M
```

6.6.3 工作原理

在前面的例子中，我们将一些简单的测试数据导入到了 Redis 实例中。随后，通过 `SAVE` 命令创建了一个 RDB 转储文件，并通过 `BGWRITEAOF` 命令重写了 AOF 文件。

通过观察这些文件的大小，我们发现数据同时以 RDB 和 AOF 的形式保存到了 Redis 的数据目录中。基于对此前案例的学习，我们很清楚地发现 AOF 文件要比 RDB 文件更大。

然后，我们通过 Vim 编辑器查看了 AOF 的开始和结束部分，其中按此前探究 AOF 文件的案例中描述的 AOF 格式保存了写入命令。

在同时启用了 RDB 和 AOF 方法后，我们将配置参数 `aof-use-rdb-preamble` 设置为 `yes`，来启用从 Redis 4.x 开始提供的新的混合持久化功能。当这个选项设置为 `yes` 后，在重写 AOF 文件时，Redis 首先会把数据集以 RDB 的格式转储到内存中并作为 AOF 文件的开始部分。在重写之后，Redis 继续使用传统的 AOF 格式在 AOF 文件中记录写入命令。你可以通过观察重写后的 AOF 文件头部和尾部清楚地了解这种混合格式。如果启用了混合持久化，那么在 AOF 文件的开头首先使用的是 RDB 格式。因为 RDB 的压缩格式可以实现更快速地重写和加载数据文件，同时也保留了 AOF 数据一致性更好的优点，所以 Redis 可以从混合持久化中获益。

6.6.4 更多细节

对于 Redis 4.x 之前的版本，我们可以同时启用 RDB 和 AOF 来实现最好的数据安全。我们建议在 Redis 4.x 之后启用 AOF 和混合持久化。请注意，启用混合持久化但却禁用 AOF，是没有任何意义的。在这种情况下，即使我们通过调用 `BGWRITEAOF` 命令手动创建 AOF 文件，也没有任何内容会被追加到 AOF 文件中。如果我们能够忍受某些可能的数据丢失，那么只使用 RDB 能够获得更好的性能。此外，根据 Redis 作者 *Antirez* 的说法，为了防止 AOF 引擎的缺陷导致丢失数据，我们并不建议只使用 AOF。

最后，Redis 保证 RDB 转储和 AOF 重写不会同时进行。当 Redis 启动时，即便 RDB 和 AOF 持久化同时启用且 AOF、RDB 文件都存在，则 Redis 总是会先加载 AOF 文件。这是因为 AOF 文件被认为能够提供更加鲁棒的数据一致性。当加载 AOF 文件时，如果启用了混合持久化，那么 Redis 将首先检查 AOF 的前五个字符。如果这五个字符是 `REDIS`（也就是 RDB 文件的魔数字符串，参见探究 RDB 文件的案例），那么该 AOF 文件就是混合格式的。Redis 服务器会先加载 RDB 部分，然后再加载 AOF 部分。

第 7 章

配置高可用和集群

在本章中，我们将学习下列案例：

- 配置 Sentinel。
- 测试 Sentinel。
- 管理 Sentinel。
- 配置 Redis Cluster。
- 测试 Redis Cluster。
- 管理 Redis Cluster。

7.1 本章概要

对于生产环境中的需求，Redis 单实例远远不能提供稳定高效、具备数据冗余和高可用（**HA, high availability**）能力的键值存储服务。使用 Redis 的主从复制和持久化可以解决数据冗余备份的问题。但是，如果没有人工干预，当主实例宕机时，整个 Redis 服务将无法恢复。尽管市面上有多种 Redis 高可用的解决方案，但 2.6 版本以后 Redis 原生支持的 Sentinel 是使用最广泛的高可用架构。利用 Sentinel，我们可以轻松地构建具备容错能力的 Redis 服务。

由于 Redis 中所存储的数据增长速度很快，一个存储了大量数据（通常 16GB 以上）的 Redis 实例的处理能力和内存容量可能会变成应用的瓶颈。随着 Redis 中数据集大小的增长，在进行持久化或主从复制时，也会越来越多地出现诸如延迟等问题。对于这种情况，水平扩展，或通过向 Redis 服务中增加更多节点（译者注：本章中的“节点”对应原文中的“node”，与前文所提到的“实例”在语义上基本等价；但由于本章的 Redis 实例处于集群的上下文语境中，因此按照惯例对两者不做

严格区分）来实现伸缩就成为了一种迫切的需要。从 3.0 版本开始支持的 Redis Cluster 正是针对这类问题提出的。Redis Cluster 是一种开箱即用的解决方案，可以将数据集通过分区的方式分布到多个 Redis 主从实例中。

在本章中，我们将按照配置、测试和管理三个步骤分别讨论如何使用 Sentinel 和 Cluster 构建满足生产环境要求的 Redis 服务。

最后，值得一提的是，作为惯例，第一个字母大写的单词 **Cluster** 特指 Redis Cluster 技术。读者可能会发现，在 Redis 3.0 发布之前，有很多第三方系统实现了 Redis 的集群功能。所以，不要将本章中讨论的 Redis Cluster 与其他使用 Redis 的数据集群系统混淆。

7.2 配置 Sentinel

顾名思义，Sentinel（哨兵）充当了 Redis 主实例和从实例的守卫者。因为单个哨兵本身也可能失效，所以一个哨兵显然不足以保证高可用。对主实例进行故障迁移的决策是基于仲裁系统的，所以至少需要三个哨兵进程才能构成一个健壮的分分布式系统来持续地监控 Redis 主实例的状态。如果有多个哨兵进程检测到主实例下线，其中的一个哨兵进程会被选举出来负责推选一个从实例替代原有的主实例。如果配置恰当，上述的整个过程都将是自动化的，无需人工干预。在本案例中，我们将演示如何配置一个由三个哨兵监控的一主两从环境。

7.2.1 准备工作

我们需要配置一个主实例和两个从实例。读者可以参考第 6 章 *Persistence* 中配置 Redis 的复制机制的案例中的步骤。在本例中，我们将在三个不同的主机上部署 Redis 服务器和 Sentinel 实例。各机器的角色、IP 地址和端口号如表 7.1 所示。

表 7.1 各机器的角色、IP 地址和端口号

角色	IP 地址	端口号
Master	192.168.0.31	6379
Slave-1	192.168.0.32	6379
Slave-2	192.168.0.33	6379
Sentinel-1	192.168.0.31	26379
Sentinel-2	192.168.0.32	26379
Sentinel-3	192.168.0.33	26379

整体的架构如图 7-1 所示。

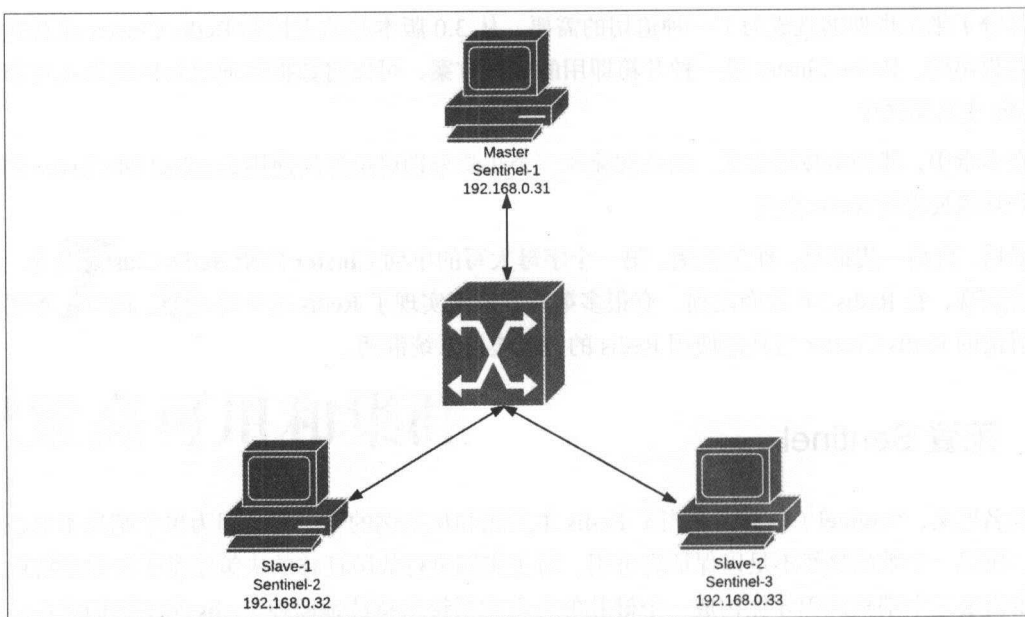


图 7.1 整体架构

这三台主机必须能够彼此通信。我们需要在 Redis 配置文件中正确地设置绑定的 IP 地址，以便让其他主机能够与 Redis 实例进行通信。默认绑定的 IP 地址是 127.0.0.1，只允许从本地访问。我们可以将要绑定的 IP 地址追加到后面，如下所示：

```
bind 127.0.0.1 192.168.0.31
```

请确认所有的 Redis 实例都已经启动并运行了。

7.2.2 操作步骤

配置 Sentinel 的步骤如下：

1. 在每台主机上准备一个配置文件 `sentinel.conf`。我们可以从源码中拷贝一个示例副本。另外，我们必须保证运行 Sentinel 实例的用户有写入配置文件的权限：

```
port 26379
dir /tmp
sentinel monitor mymaster 192.168.0.31 6379 2
sentinel down-after-milliseconds mymaster 30000
sentinel parallel-syncs mymaster 1
sentinel failover-timeout mymaster 180000
```

2. 在三台主机上启动 Sentinel 进程。

```
user@192.168.0.31:~$bin/redis-server conf/sentinel.conf --sentinel
user@192.168.0.32:~$bin/redis-server conf/sentinel.conf --sentinel
user@192.168.0.33:~$bin/redis-server conf/sentinel.conf --sentinel
```

3. 查看 Sentinel-1 的日志。

```
21758:X 29 Oct 22:31:51.001 # Sentinel ID is 3
ef95f7fd6420bfe22e38bfdded1399382a63ce5b
21758:X 29 Oct 22:31:51.001 # +monitor master mymaster 192.168.0.31 6379
quorum 2
21758:X 29 Oct 22:31:51.001 * +slave slave 192.168.0.32:6379 192.168.0.32 6379
@ mymaster 192.168.0.31 6379
21758:X 29 Oct 22:31:51.003 * +slave slave 192.168.0.33:6379 192.168.0.33 6379
@ mymaster 192.168.0.31 6379
21758:X 29 Oct 22:31:52.021 * +sentinel sentinel
d24979c27871eafa62e797dlc8e5lacc99bbda72 192.168.0.32 26379 @ mymaster
192.168.0.31 6379
21758:X 29 Oct 22:32:17.241 * +sentinel sentinel
a276b044b26100570bb1a4d83d5b3f9d66729f64 192.168.0.33 26379 @ mymaster
192.168.0.31 6379
```

4. 使用 redis-cli 连接到 Sentinel-1 并执行 INFO SENTINEL 命令。请不要忘记指定端口，26379。

```
user@192.168.0.31:~$ bin/redis-cli -p 26379
127.0.0.1:26379> INFO SENTINEL
# Sentinel
sentinel_masters:1
sentinel_tilt:0
sentinel_running_scripts:0
sentinel_scripts_queue_length:0
sentinel_simulate_failure_flags:0
master0:name=mymaster,status=ok,address=192.168.0.31:6379,slaves=2,sentinels=3
```

5. 在 Sentinel-1 上查看 sentinel.conf 的内容。

```
user@192.168.0.31:~$ cat conf/sentinel.conf
```



```
...
# Generated by CONFIG REWRITE
sentinel known-slave mymaster 192.168.0.33 6379
sentinel known-slave mymaster 192.168.0.32 6379
sentinel known-sentinel mymaster 192.168.0.33 26379
    a276b044b26100570bb1a4d83d5b3f9d66729f64
sentinel known-sentinel mymaster 192.168.0.32 26379
    d24979c27871eafa62e797d1c8e51acc99bbda72
sentinel current-epoch 0
```

7.2.3 工作原理

在第 1 步中，我们为 Sentinel 进程准备了一个配置文件。由于在 Redis 源码包中有一个示例文件 `sentinel.conf`，所以在本例子中我们只是基于该文件做了一些修改。如前所述，因为 Sentinel 是 Redis 实例的守护进程，因此它必须监听在与 Redis 实例不同的端口上。Sentinel 的默认端口号是 26379。如果要将一个新的主实例添加到 Sentinel 中进行监控，那么我们可以按如下格式在配置文件中增加一行：`sentinel monitor <master-name> <ip> <port> <quorum>`。

在本例中，`sentinel monitor mymaster 192.168.0.31 6379 2` 表示 Sentinel 将监控 192.168.0.31:6379 的主实例，该主实例名为 `mymaster`。`<quorum>` 指在采取故障迁移操作前，发现并同意主实例不可达的最少哨兵数。`down-after-milliseconds` 选项指在标记实例下线前不可达的最长毫秒数。Sentinel 每秒钟都向实例发送 PING 命令来检查其是否可达。

在本例中，如果某个实例超过 30 秒仍未响应 ping 命令，那么它将被视做下线。当主实例发生故障迁移时，其中的一个从实例将被选为新的主实例，而其他的从实例则需要从新的主实例那里进行主从复制。`parallel-syncs` 参数表示有几个从实例可以同时从新的主实例进行数据同步。

我们可以通过 `redis-server <sentinel.conf> --sentinel` 启动一个 Sentinel 进程。如果我们是源码编译 Redis 的，那么会找到一个 `redis-sentinel` 文件（指向 `redis-server` 的符号链接）。Sentinel 进程也可以通过 `redis-sentinel <sentinel.conf>` 启动。

从第 3 步的日志和第 4 步中 INFO SENTINEL 输出的最后一行，我们可以看到 Sentinel-1 成功地检测到了从实例和其他哨兵（如果查看 Sentinel-2 和 Sentinel-3 的日志会发现相同的内容）。

读者可能想知道，Sentinel 进程是如何检测到从服务器和其他哨兵的（毕竟我们只在配置文件中指定了主实例的信息）。我们可以认为，为了获得从实例的信息，Sentinel 会向主实例发送 INFO REPLICATION 命令。即使从实例的结构有多级，也可以通过递归的方式找到它们。实际上，每个 Sentinel 进程每 10 秒钟会向其监控的所有 Redis 实例（包括主实例和被检测到的从实例）发送 INFO REPLICATION 命令，来获取整个主从复制拓扑结构的最新信息。

为了探测其他哨兵及与其他哨兵进行通信,每个 Sentinel 进程每两秒钟会向一个名为 `__sentinel_1__:hello` 的频道发布一条消息,报告其自身及所监控主实例的状态。因此,只要订阅该频道就能发现其他哨兵的信息了。

通过连接到任意一个 Redis 实例并订阅该频道即可查看该频道中的消息。

```
user@redis-master:/redis$ bin/redis-cli -h 192.168.0.31
192.168.0.31:6379> SUBSCRIBE __sentinel__:hello
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "__sentinel__:hello"
3) (integer) 1
1) "message"
2) "__sentinel__:hello"
3) "192.168.0.31,26379,3ef95f7fd6420bfe22e38bfded1399382a63ce5b,0,mymaster
,192.1
68.0.31,6379,0"
1) "message"
2) "__sentinel__:hello"
...
"192.168.0.31,26379,3ef95f7fd6420bfe22e38bfded1399382a63ce5b,0,mymaster,192.1
68.0.31,6379,0"
```

当从实例和其他哨兵的信息发生改变时, Sentinel 的配置文件也将被更新。这就是 Sentinel 进程必须对配置文件有写入权限的原因。

7.2.4 更多细节

值得一提的是,使用 Redis Sentinel 需要客户端的支持。当我们在第 4 章使用 Redis 进行开发中学习 Redis 的 Java/Python 客户端时,我们将主实例的地址传给了 API。但是,在启用了 Redis Sentinel 之后,主实例的地址在主实例发生故障迁移时会发生改变。要获取最新的主实例的信息,客户端就需要从 Sentinel 进行查询。这可以通过 `sentinel get-master-addr-by-name <master-name>` 命令实现。实际上,上述过程更加复杂,因而我们不打算在本书中进行讨论。幸运的是, Jedis 和 `redis-py` 库都支持 Sentinel。

7.2.5 相关内容

- Redis Sentinel 的官方文档,请参阅: <https://redis.io/topics/sentinel>。

7.3 测试 Sentinel

在之前的案例中，我们演示了如何配置由三个哨兵监控的一主两从的环境。在本案例中，我们将在这个环境中进行一些实验，并验证哨兵们的工作是否正常。我们还将详细解释主实例故障迁移的过程。

7.3.1 准备工作

我们必须完成本章配置 *Sentinel* 一节中的配置。此外，我们还可以参考前面配置 *Sentinel* 的案例中准备工作小节所示的表格。

7.3.2 操作步骤

接下来，让我们按照以下的步骤测试前一案例中配置的 Redis Sentinel：

1. 手动触发主实例故障迁移：

（1）使用 `redis-cli` 连接到其中一个哨兵；这里我们连接到 Sentinel-2（192.168.0.32）：

```
192.168.0.32:26379> SENTINEL FAILOVER MYMASTER
OK
```

（2）验证原来的主实例 192.168.0.31 已经进行了故障转移，现在变成了从实例：

```
192.168.0.31:26379> INFO REPLICATION
# Replication
role:slave
master_host:192.168.0.33
master_port:6379
master_link_status:up
...
```

（3）检查 Sentinel-2 的日志：

```
2283:X 12 Nov 15:35:14.782 # Executing user requested FAILOVER of 'mymaster'
2283:X 12 Nov 15:35:14.782 # +new-epoch 1
2283:X 12 Nov 15:35:14.782 # +try-failover master mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:14.789 # +vote-for-leader
d24979c27871eafa62e797d1c8e51acc99bbda72 1
```



```

2283:X 12 Nov 15:35:14.789 # +elected-leader master mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:14.789 # +failover-state-select-slave master mymaster
192.168.0.31 6379
2283:X 12 Nov 15:35:14.872 # +selected-slave slave 192.168.0.33:6379
192.168.0.33 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:14.872 * +failover-state-send-slaveof-noone slave
192.168.0.33:6379 192.168.0.33 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:14.949 * +failover-state-wait-promotion slave
192.168.0.33:6379 192.168.0.33 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:15.799 # +promoted-slave slave 192.168.0.33:6379
192.168.0.33 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:15.800 # +failover-state-reconf-slaves master mymaster
192.168.0.31 6379
2283:X 12 Nov 15:35:15.852 * +slave-reconf-sent slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:16.503 * +slave-reconf-inprog slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:16.503 * +slave-reconf-done slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:16.580 # +failover-end master mymaster 192.168.0.31 6379
2283:X 12 Nov 15:35:16.580 # +switch-master mymaster 192.168.0.31 6379
192.168.0.33 6379
2283:X 12 Nov 15:35:16.580 * +slave slave 192.168.0.32:6379 192.168.0.32 6379
@ mymaster 192.168.0.33 6379
2283:X 12 Nov 15:35:16.581 * +slave slave 192.168.0.31:6379 192.168.0.31 6379
@ mymaster 192.168.0.33 6379

```

(4) 在 192.168.0.33 (新主实例) 上检查 redis-server 的日志:

```

2274:M 12 Nov 15:35:14.953 # Setting secondary replication ID to 8
a005b14ac7166dfc913846060bee4a980f97785, valid up to offset: 92256. New
replication ID is a897d63fb211d7ebf6c1269998dab1779d14f8a4
2274:M 12 Nov 15:35:14.953 # Connection with master lost.
2274:M 12 Nov 15:35:14.953 * Caching the disconnected master state.
2274:M 12 Nov 15:35:14.953 * Discarding previously cached master state.
2274:M 12 Nov 15:35:14.953 * MASTER MODE enabled (user request from 'id=3 addr
=192.168.0.32:60540 fd=9 name=sentinel-d24979c2-cmd age=356 idle=0 flags=x
db=0 sub=0 psub=0 multi=3 qbuf=0 qbuf-free=32768 obl=36 oll=0 omem=0

```

```
events=r cmd=exec')
2274:M 12 Nov 15:35:14.954 # CONFIG REWRITE executed with success.
2274:M 12 Nov 15:35:16.452 * Slave 192.168.0.32:6379 asks for synchronization
2274:M 12 Nov 15:35:16.452 * Partial resynchronization not accepted: Requested
offset for second ID was 92534, but I can reply up to 92256
...
2274:M 12 Nov 15:35:16.462 * Synchronization with slave 192.168.0.32:6379
succeeded
2274:M 12 Nov 15:35:26.839 * Slave 192.168.0.31:6379 asks for synchronization
...
2274:M 12 Nov 15:35:26.936 * Background saving terminated with success
2274:M 12 Nov 15:35:26.937 * Synchronization with slave 192.168.0.31:6379
succeeded
```

(5) 检查 Sentinel-1 配置文件 sentinel.conf 的内容：

```
user@192.168.0.31:~$ cat conf/sentinel.conf
port 26379
dir "/tmp"
sentinel myid d24979c27871eafa62e797d1c8e51acc99bbda72
sentinel monitor mymaster 192.168.0.33 6379 2
...
```

2. 模拟主实例下线。

现在的主实例是 192.168.0.33。让我们关闭这台机器上的 Redis 实例，看看哨兵会做什么动作：

(1) 通过 redis-cli 连接到 192.168.0.33 上的 Redis 实例并将其关闭：

```
192.168.0.33:6379> SHUTDOWN
not connected>
```

(2) 检查 192.168.0.31 和 192.168.0.32 的状态：

```
192.168.0.31:6379> INFO REPLICATION
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.0.32,port=6379,state=online,offset=349140,lag=1
```

```
192.168.0.32:6379> INFO REPLICATION
```

```
# Replication
```

```
role:slave
```

```
master_host:192.168.0.31
```

```
master_port:6379
```

```
master_link_status:up
```

我们可以看到，192.168.0.31 被提升成了新的主实例。

（3）检查三个哨兵的日志。

```
Sentinel-1 (3ef95f7fd6420bfe22e38bfded1399382a63ce5b):
```

```
2931:X 12 Nov 17:05:02.446 # +sdown master mymaster 192.168.0.33 6379
```

```
2931:X 12 Nov 17:05:03.570 # +odown master mymaster 192.168.0.33 6379 #quorum
```

```
2/2
```

```
2931:X 12 Nov 17:05:03.570 # +new-epoch 2
```

```
2931:X 12 Nov 17:05:03.570 # +try-failover master mymaster 192.168.0.33 6379
```

```
2931:X 12 Nov 17:05:03.573 # +vote-for-leader 3
```

```
ef95f7fd6420bfe22e38bfded1399382a63ce5b 2
```

```
2931:X 12 Nov 17:05:03.573 # a276b044b26100570bb1a4d83d5b3f9d66729f64 voted
```

```
for d24979c27871eafa62e797d1c8e51acc99bbda72 2
```

```
2931:X 12 Nov 17:05:04.224 # +config-update-from sentinel
```

```
d24979c27871eafa62e797d1c8e51acc99bbda72 192.168.0.32 26379 @ mymaster  
192.168.0.33 6379
```

```
2931:X 12 Nov 17:05:04.224 # +switch-master mymaster 192.168.0.33 6379
```

```
192.168.0.31 6379
```

```
...
```

```
2931:X 12 Nov 17:05:34.283 # +sdown slave 192.168.0.33:6379 192.168.0.33 6379
```

```
@ mymaster 192.168.0.31 6379
```

```
Sentinel-2 (d24979c27871eafa62e797d1c8e51acc99bbda72):
```

```
3055:X 12 Nov 17:05:02.394 # +sdown master mymaster 192.168.0.33 6379
```

```
3055:X 12 Nov 17:05:03.505 # +odown master mymaster 192.168.0.33 6379 #quorum
```

```
2/2
```

```
3055:X 12 Nov 17:05:03.505 # +new-epoch 2
```

```
3055:X 12 Nov 17:05:03.505 # +try-failover master mymaster 192.168.0.33 6379
```

```
3055:X 12 Nov 17:05:03.507 # +vote-for-leader
```

```
d24979c27871eafa62e797d1c8e51acc99bbda72 2
```

```
3055:X 12 Nov 17:05:03.516 # a276b044b26100570bb1a4d83d5b3f9d66729f64 voted
```

```
for d24979c27871eafa62e797d1c8e51acc99bbda72 2
```



```

3055:X 12 Nov 17:05:03.584 # +elected-leader master mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:03.584 # +failover-state-select-slave master mymaster
192.168.0.33 6379
3055:X 12 Nov 17:05:03.668 # +selected-slave slave 192.168.0.31:6379
192.168.0.31 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:03.668 * +failover-state-send-slaveof-noone slave
192.168.0.31:6379 192.168.0.31 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:03.758 * +failover-state-wait-promotion slave
192.168.0.31:6379 192.168.0.31 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:04.135 # +promoted-slave slave 192.168.0.31:6379
192.168.0.31 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:04.135 # +failover-state-reconf-slaves master mymaster
192.168.0.33 6379
3055:X 12 Nov 17:05:04.224 * +slave-reconf-sent slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:04.609 # -odown master mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:05.147 * +slave-reconf-inprog slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:05.147 * +slave-reconf-done slave 192.168.0.32:6379
192.168.0.32 6379 @ mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:05.201 # +failover-end master mymaster 192.168.0.33 6379
3055:X 12 Nov 17:05:05.201 # +switch-master mymaster 192.168.0.33 6379
192.168.0.31 6379
3055:X 12 Nov 17:05:05.201 * +slave slave 192.168.0.32:6379 192.168.0.32 6379
@ mymaster 192.168.0.31 6379
3055:X 12 Nov 17:05:05.201 * +slave slave 192.168.0.33:6379 192.168.0.33 6379
@ mymaster 192.168.0.31 6379
3055:X 12 Nov 17:05:35.282 # +sdown slave 192.168.0.33:6379 192.168.0.33 6379
@ mymaster 192.168.0.31 6379
Sentinel-3 (a276b044b26100570bb1a4d83d5b3f9d66729f64):
2810:X 12 Nov 17:05:02.519 # +sdown master mymaster 192.168.0.33 6379
2810:X 12 Nov 17:05:03.512 # +new-epoch 2
2810:X 12 Nov 17:05:03.517 # +vote-for-leader
d24979c27871eafa62e797d1c8e51acc99bbda72 2
2810:X 12 Nov 17:05:04.225 # +config-update-from sentinel
d24979c27871eafa62e797d1c8e51acc99bbda72 192.168.0.32 26379 @ mymaster
192.168.0.33 6379

```

```
2810:X 12 Nov 17:05:04.225 # +switch-master mymaster 192.168.0.33 6379
192.168.0.31 6379
2810:X 12 Nov 17:05:04.225 * +slave slave 192.168.0.32:6379 192.168.0.32 6379
@mymaster 192.168.0.31 6379
2810:X 12 Nov 17:05:04.225 * +slave slave 192.168.0.33:6379 192.168.0.33 6379
@mymaster 192.168.0.31 6379
2810:X 12 Nov 17:05:34.277 # +sdown slave 192.168.0.33:6379 192.168.0.33 6379
@mymaster 192.168.0.31 6379
```

3. 模拟两个从实例下线。

由于我们已经关闭了 192.168.0.33 上的 Redis 实例，目前剩下一个主实例（192.168.0.31）和一个从实例（192.168.0.32）：

（1）将主实例上的 min-slaves-to-write 设为 1：

```
192.168.0.31:6379> CONFIG SET MIN-SLAVES-TO-WRITE 1
OK
```

（2）关闭剩下的最后一个从实例 192.168.0.32：

```
192.168.0.32:6379> SHUTDOWN
```

（3）尝试向主实例中写入数据：

```
127.0.0.1:6379> SET test_key 12345
(error) NOREPLICAS Not enough good slaves to write.
```

4. 模拟一个哨兵下线。

（1）在进行这个实验之前，让我们重新启动 192.168.0.32 和 192.168.0.33 上的 Redis 实例：

```
user@192.168.0.32:/redis$ bin/redis-server conf/redis.conf
user@192.168.0.33:/redis$ bin/redis-server conf/redis.conf
```

（2）停止 Sentinel-1：

```
192.168.0.31:26379> SHUTDOWN
```

（3）关闭主实例 192.168.0.31，并验证是否发生了故障迁移：

```

192.168.0.31:6379> SHUTDOWN
192.168.0.32:6379> info replication
# Replication
role:master
connected_slaves:1
slave0:ip=192.168.0.33,port=6379,state=online,offset=782227,lag=0
192.168.0.33:6379> info replication
# Replication
role:slave
master_host:192.168.0.32
master_port:6379
master_link_status:up

```

5. 模拟两个哨兵下线。

（1）重启 192.168.0.31 上的 Redis 实例。当前的主实例为 192.168.0.32：

```
user@192.168.0.31:/redis$ bin/redis-server conf/redis.conf
```

（2）停止 Sentinel-2，并关闭主实例 192.168.0.32：

```
192.168.0.32:6379> SHUTDOWN
```

（3）检查 Sentinel-3 的日志：

```

...
2810:X 12 Nov 18:22:41.171 # +sdown master mymaster 192.168.0.32 6379

```

主实例没有发生故障迁移。

7.3.3 工作原理

下面，让我们逐一解释前一节中所发生的现象。

手动触发主实例故障迁移

在该实验中，我们手动地强制哨兵进行了主实例的故障迁移并重新选出了一个从实例。这是通过在 Sentinel-2 上执行 `sentinel failover <master-name>` 命令完成的。我们可以看到，192.168.0.33 被选举为新的主实例，而原来的主实例变成了从实例。

现在，让我们通过 Sentinel-2 的日志详细了解一下上述的过程；在日志中记录了多个 Sentinel 事件（+vote-for-leader、+elected-leader、+selected-slave，等等）。这些事件的名字大多数都很直白，我们可以在更多细节一节中找到包含所有 Sentinel 事件的表格。

1. 由于故障迁移是手动触发的，所以在执行故障迁移操作之前，Sentinel 不需要寻求其他哨兵的同意。Sentinel-2 无需任何共识就直接被选举为 leader。

2. 接下来，Sentinel 挑选了一个从实例将其提成为主实例，也就是本实验中的 192.168.0.33。

3. Sentinel 向 192.168.0.33 发送 `slaveof no one` 命令使之变成主实例。如果我们检查 192.168.0.33 的日志，会发现服务器收到了来自 Sentinel-2 的命令，停止从老的主实例 192.168.0.31 上进行主从复制，并被提升为一个主实例。

4. Sentinel 重新配置老的主实例 192.168.0.31 和另一个从实例 192.168.0.32，让它们从新的主实例那里进行主从复制。

5. 在最后一步中，Sentinel 更新新主实例的信息，并通过频道 `__sentinel__:hello` 向其他哨兵广播这些信息，从而让所有客户端获得新主实例的信息。

配置文件 `redis.conf` 和 `sentinel.conf` 也相应地进行了更新，以匹配新主实例的角色。

模拟主实例下线

在该实验中，我们通过手动关闭的方式模拟了主实例 192.168.0.33 的下线。哨兵将 192.168.0.31 提升为新的主实例并完成了故障迁移。

让我们看看整个过程是如何进行的：

1. Sentinel-1 于 17:05:02.446 发现主实例不可达。正如我们在此前的案例中所提到的，每一个哨兵都将定期地向主实例、从实例及其他哨兵发送 `ping` 命令。如果 `ping` 命令超时（指定时间内未收到响应），对应的服务器将被视为下线。不过，这仅仅是单个哨兵的主观看法，即主观下线（subjectively down，Sentinel 的 +sdown 事件）。在本例中，Sentinel-1 将主实例标记为 +sdown（主观下线）。

2. 为了防止虚假警报，标记主实例 +sdown 的哨兵会向其他哨兵发送请求，要求他们检查主实例的状态。只有当多于 <quorum> 个哨兵认为主实例下线时，才会发生故障迁移，这被称为客观下线（objectively down，+odown）。在本例中，Sentinel-1 在 17:05:03.570 得到了其他哨兵的响应，并将主实例标记为 +odown（客观下线）。

3. 接下来，Sentinel-1 尝试执行故障迁移，但没有被选为 leader。

4. 几乎在同一时间，Sentinel-2 也将主实例标记为 +sdown 和 +odown，且被选为执行故障迁移的 leader。故障转移的后续过程与手动触发主实例故障迁移中的步骤相同。

leader 是如何被选出来的呢？当一个哨兵标记 +odown 后，投票即开始；哨兵会开始从其他哨兵那里征集选票。每个哨兵只有一票。当另一个哨兵收到拉票请求时，如果它以前还没有投过票，则

接受请求并回复拉票的哨兵；否则，则拒绝请求，并向其回复“刚刚已经把票投给了其他哨兵”。如果某个哨兵得到了大于等于最大值（quorum，哨兵数/2+1）张选票（包括它自己；哨兵在从其他哨兵处拉票前会先投票给自己），那么这个哨兵将成为 leader。如果没有 leader 当选，则重新进行上述过程。

让我们回到哨兵的日志。Sentinel-1 在 17:05:03.573 发出了拉票请求，却收到了 Sentinel-3 (id: a276b044b26100570bb1a4d83d5b3f9d66729f64) 已经把票投给了 Sentinel-2 (id: d24979c27871eafa62e797d1c8e51acc99bbda72) 的响应。Sentinel-2 在更早的 17:05:03.507 发出了拉票请求，并得到了 Sentinel-3 同意投票给 Sentinel-2 的响应。因此，Sentinel-2 获得了两票并成为了 leader。

模拟两个从实例下线

在该实验中，我们将 min-slaves-to-write 设置 1，并关闭了两个从实例，只留下主实例运行。min-slaves-to-write 参数表示接受写入请求所需的最少从实例数。由于没有从实例，所以写入请求被主实例拒绝了。

模拟一个哨兵下线

在该实验中，停止一个哨兵并不影响故障迁移过程，因为 quorum 对于客观下线和 leader 选举仍然是有效的。

模拟两个哨兵下线

只剩下一个单独的哨兵时将不能触发客观下线和 leader 选举。因此，哨兵只是将主实例标记为 +sdown 而已，并未进行故障迁移。

7.3.4 更多细节

在 Redis Sentinel 中有许多类型的事件。由于本书篇幅的限制，我们无法讲解全部的事件类型。读者可以参阅 Redis Sentinel 的文档了解更多细节：<https://redis.io/topics/sentinel>。

7.4 管理 Sentinel

我们在前两个案例中学习了如何配置和测试 Redis Sentinel。除了监控主实例和从实例的状态外，Redis Sentinel 还提供了一些便捷的功能，比如当 Sentinel 事件或故障迁移发生时执行脚本。在本案例中，我们将首先介绍一些常用的 Sentinel 命令，然后学习如何使用脚本功能来自动化一些常见操作。

7.4.1 准备工作

我们必须完成本章配置 *Sentinel* 案例中的配置步骤，并且让 Redis 实例和哨兵运行起来。

7.4.2 操作步骤

接下来，让我们按照以下的步骤来学习如何管理 Sentinel：

1. 学习哨兵的命令。

（1）使用 `redis-cli` 连接到其中一个哨兵：

```
user@192.168.0.33:~$bin/redis-cli -h 192.168.0.33 -p 26379
192.168.0.33:26379>
```

（2）使用 `SENTINEL GET-MASTER-ADDR-BY-NAME <master-name>` 命令获取当前主实例的信息：

```
192.168.0.33:26379> SENTINEL GET-MASTER-ADDR-BY-NAME mymaster
1) "192.168.0.31"
2) "6379"
```

（3）使用 `SENTINEL MASTERS` 命令获取所有被监控主实例的状态：

```
192.168.0.33:26379> SENTINEL MASTERS
1) 1) "name"
   2) "mymaster"
   3) "ip"
   4) "192.168.0.31"
   5) "port"
   ...
  17) "last-ok-ping-reply"
  18) "364"
   ...
```

（4）类似地，使用 `SENTINEL SLAVES <master-name>` 命令获取一个被监控主实例的所有从实例的信息：

```
192.168.0.33:26379> SENTINEL SLAVES mymaster
1) 1) "name"
```



```
2) "192.168.0.33:6379"
3) "ip"
4) "192.168.0.33"
5) "port"
6) "6379"
7) "runid"
8) "23b3730d1b32fde674c5ea07b9440c08cee9fabe"
...
```

(5) 使用 SENTINEL SET 命令更新哨兵的配置：

```
192.168.0.33:26379> SENTINEL SET MYMASTER DOWN-AFTER-MILLISECONDS 1000
OK
```

2. 在 Sentinel 事件发生时执行脚本。

(1) 假设我们希望当 Sentinel 事件（例如，+sdown、+odown）发生时自动发送电子邮件。这里，我们使用的是 Python 脚本。读者可以在本书附带的源码包中找到这个 Python 脚本。这里提供的脚本只是一个示例；在真实环境中，我们需要设置正确的 SMTP 服务器和登录凭证。

(2) 在其中一个哨兵上使用 SENTINEL SET 命令更新通知脚本的配置。例如，让我们在 Sentinel-3（192.168.0.33）上启用该功能：

```
192.168.0.33:26379> SENTINEL SET mymaster notification-script mymaster /redis/
scripts/sentinel_events_notify.py
OK
```

每当有 Sentinel 事件发生时，就会触发脚本 /redis/scripts/sentinel_events_notify.py。

3. 在故障迁移发生时执行脚本。

我们可以配置 Redis Sentinel 在故障迁移发生时自动地执行一个脚本。当主实例和从实例的配置不同时，这个功能非常有用。例如，我们可能希望当主实例故障迁移发生时禁用主实例上的 RDB 持久化选项，而在从实例上保持启用。当主实例发生故障迁移时，一个从实例将被提升为主实例，但这个从实例的配置不能被哨兵更新；因此，除非我们手动关闭，否则新主实例上的 RDB 持久化选项仍将启用。通过这个功能，我们可以创建一个脚本来自动地在实例角色发生改变时更新配置。

(1) 禁用主实例上的 RDB 持久化（192.168.0.31）：

```
127.0.0.1:6379> CONFIG SET SAVE ""
OK
```

(2) 准备一个在故障迁移时根据当前角色更新 RDB 配置的脚本。读者可以在本书附带的源码包中找到这个 Bash shell 脚本。

(3) 更新所有哨兵的配置：

```
192.168.0.31:26379> sentinel set mymaster client-reconfig-script /redis/
scripts/rdb_control.bash
OK
192.168.0.32:26379> sentinel set mymaster client-reconfig-script /redis/
scripts/rdb_control.bash
OK
192.168.0.33:26379> sentinel set mymaster client-reconfig-script /redis/
scripts/rdb_control.bash
OK
```

(4) 触发一次故障迁移：

```
192.168.0.32:26379> SENTINEL FAILOVER mymaster
OK
```

(5) 验证新主实例（192.168.0.33）的 RDB 持久化选项是禁用的，而老主实例（192.168.0.31）的 RDB 持久化选项是启用的：

```
192.168.0.33:6379> INFO REPLICATION
# Replication
role:master
connected_slaves:2
...
192.168.0.33:6379> CONFIG GET save
1) "save"
2) ""
192.168.0.31:6379> CONFIG GET save
1) "save"
2) "900 1 300 10 60 10000"
```

7.4.3 工作原理

在 Sentinel 事件发生时执行脚本的示例中，我们配置了一个 Python 脚本来在新的 Sentinel 事件发生时自动地发送通知邮件。这个脚本在 Sentinel 配置中称为 `notification-script`。传递给脚本

的参数是 `<event_type>` 和 `<event_description>`。这个功能经常被用来向管理员或系统运维人员通知某些关键事件。我们可能会希望在脚本中添加过滤器或设置通知的级别，以实现只通知特定的事件。

在故障迁移发生时执行脚本的示例中，我们配置了一个 `shell` 脚本来在发生故障迁移时自动地更新 RDB 持久化选项。这个脚本在 Sentinel 配置中称为 `client-reconfig-script`。

传递给脚本的参数是 `<master-name>` `<role>` `<state>` `<from-ip>` `<from-port>` `<to-ip>` `<to-port>`，其中 `<state>` 永远是 `failover`，`<role>` 是当前哨兵的角色（`leader` 或 `observer`），`<from-ip>` 和 `<from-port>` 是旧主实例的 IP 地址和端口，`<to-ip>` 和 `<to-port>` 则是新主实例的 IP 地址和端口。

在脚本中，我们首先通过检查 IP 地址判断当前实例的角色。请注意，这是故障迁移后的新角色。然后，根据角色，调用 `redis-cli` 来更新 RDB 持久化选项，当角色是从实例时启用，当角色是主实例时禁用。

7.4.4 更多细节

在配置中启用了 `notification-script` 和 `client-config-script` 选项的哨兵都将执行脚本。对于上面电子邮件通知的示例而言，因为我们感兴趣的事件通常会出现所有的哨兵上且只应发送一封电子邮件，所以只需要在一个哨兵上启用 `notification-script` 即可。

如果执行成功，脚本应该返回 0。如果返回值为 1，那么脚本将被重试 10 次。如果一个脚本在 60 秒内还没有执行完，那么它将被 `SIGKILL` 信号结束并被重试 10 次。如果脚本的返回值大于 1，那么脚本将不会被重试。

7.5 配置 Redis Cluster

在之前的案例中，我们已经学习了如何使用 Redis Sentinel 配置、测试和维护一个高可用的架构。正如我们在本章概要中所提到的，当 Redis 中的数据量急剧增长时，必须对其进行分区。毫无疑问，对于这个场景，从 Redis 3.0 版本开始支持的 Redis Cluster（译者注：下称 Redis 集群）是非常有用的。在本案例中，我们将按照配置、测试和维护的步骤学习如何使用 Redis 集群实现自动的数据分片（`sharding`）和高可用。在本案例中，我们将首先学习如何配置一个 Redis 集群，并讨论 Redis 集群是如何工作的。

7.5.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载并安装 Redis 的案例中所描述的步骤安装一个 Redis 服务器。

为了更好地理解 Redis 集群的工作方式，我们需要对 Redis 主从复制具备基本的认识，也就是本书第 5 章复制一章的主题。

为了准备环境，我们需要将源码 `src/` 目录中的 `redis-trib.rb` 脚本复制到 `script` 文件夹中。

7.5.2 操作步骤

在本节中，我们将配置一个 Redis 集群，该集群包括三个主节点，每个主节点都有一个从节点。此集群的拓扑结构如图 7.2 所示。

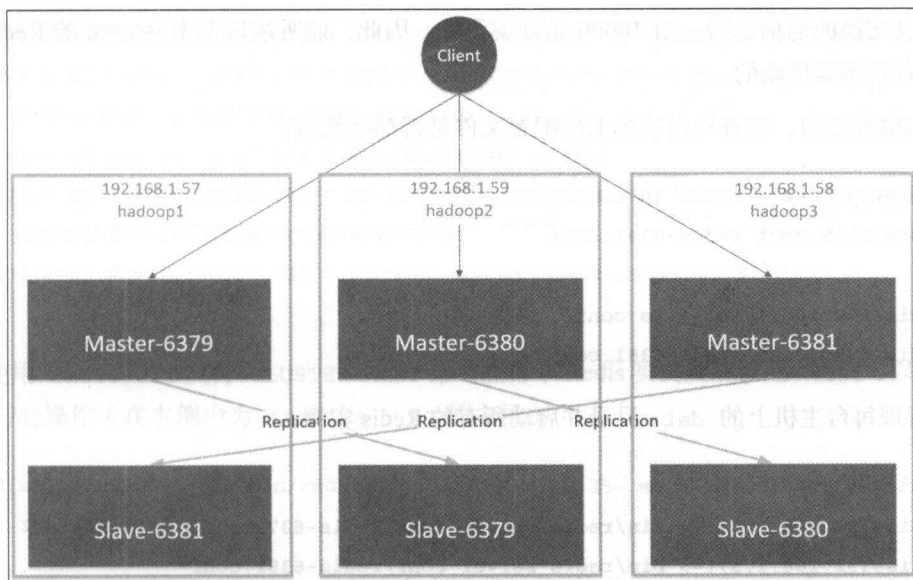


图 7.2 集群的拓扑结构

1. 每个 Redis 实例都有自己的配置文件 (`redis.conf`)。启用集群功能需要给每个 Redis 实例准备一个配置文件，然后相应地更改 IP、监听端口和 `log` 文件路径（限于本书的篇幅，我们只展示了第一台主机上其中一个实例的配置文件，其余配置文件读者可以根据自己的环境对目录和监听端口进行规划修改）：

```
redis@192.168.1.57:~> cat conf/redis-6379.conf
daemonize yes
pidfile "/redis/run/redis-6379.pid"
port 6379
bind 192.168.1.57
logfile "/redis/log/redis-6379.log"
```

```
dbfilename "dump-6379.rdb"
dir "/redis/data"
...
cluster-enabled yes
cluster-config-file nodes-6379.conf
cluster-node-timeout 10000
```

注意

当 Redis 集群运行时, 每个节点会打开两个 TCP 套接字。第一个套接字是用于客户端连接的标准 Redis 通信协议; 第二个套接字的端口号是第一个端口号加上 10000, 被用作实例间信息交换的通信总线。值 10000 是硬编码的。因此, 监听端口大于 55536 的 Redis 集群节点是不能启动的。

2. 在继续之前, 检查每台主机上的配置文件是否都已就绪:

```
redis@192.168.1.57:~> ls conf/
redis-6379.conf redis-6381.conf
...
redis@192.168.1.58:~> ls conf/
redis-6380.conf redis-6381.conf
```

3. 清理每台主机上的 data 目录并启动所有的 Redis 实例:

```
redis@192.168.1.57:~> rm -rf data/*
redis@192.168.1.57:~> bin/redis-server conf/redis-6379.conf
redis@192.168.1.57:~> bin/redis-server conf/redis-6381.conf
...
redis@192.168.1.58:~> rm -rf data/*
redis@192.168.1.58:~> bin/redis-server conf/redis-6380.conf
redis@192.168.1.58:~> bin/redis-server conf/redis-6381.conf
```

4. Redis 实例启动后, 将会在 data 目录下生成节点的配置文件。打开其中的一个, 看看里面有什么:

```
redis@192.168.1.57:~> cat data/nodes-6379.conf
58285fa03c19f6e6f633fb5c58c6a314bf25503f :0@0 myself,master - 0 0 0 connected
vars currentEpoch 0 lastVoteEpoch 0
```

5. 在配置集群前, 我们通过 INFO CLUSTER 命令获取 Redis 集群的运行信息, 并使用操作系统的 ps 命令列出 Redis 进程 (为了简明起见, 只显示了主实例 192.168.1.57)。

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 INFO CLUSTER
# Cluster
cluster_enabled:1
redis@192.168.1.57:~> ps -ef |grep redis-server
redis 119911 1 0 16:22 ? 00:00:00 bin/redis-server 192.168.1.57:6379 [cluster]
redis 119942 1 0 16:22 ? 00:00:00 bin/redis-server 192.168.1.57:6381 [cluster]
```

6. 检查实例的日志：

```
redis@192.168.1.57:~> vim log/redis-6379.log
...
26569:C 05 Nov 16:50:33.832 # oO0OoO0OoO0Oo Redis is starting oO0OoO0OoO0Oo
26569:C 05 Nov 16:50:33.832 # Redis version=4.0.1, bits=64, commit=00000000,
modified=0, pid=26569, just started
26569:C 05 Nov 16:50:33.832 # Configuration loaded
26570:M 05 Nov 16:50:33.835 * No cluster configuration found, I'm 58285
fa03c19f6e6f633fb5c58c6a314bf25503f
26570:M 05 Nov 16:50:33.839 * Running mode=cluster, port=6379.
...
```

7. 使用 `redis-cli` 执行 `CLUSTER MEET` 命令让每个 Redis 实例发现彼此。我们可以只在一台主机上进行此操作（在本例中为 192.168.1.57 的主机）：

```
redis@192.168.1.58:~> bin/redis-cli -h 192.168.1.57 -p 6379 CLUSTER MEET
192.168.1.57 6379
OK
...
redis@192.168.1.58:~> bin/redis-cli -h 192.168.1.57 -p 6379 CLUSTER MEET
192.168.1.58 6380
OK
redis@192.168.1.58:~> bin/redis-cli -h 192.168.1.57 -p 6379 CLUSTER MEET
192.168.1.58 6381
OK
```

注意

如果我们把主机名当成了集群中节点的地址，那么即使主机名能够正确地映射为 IP 地址，也会报如下的错误：

```
redis@192.168.1.58:~> bin/redis-cli -h 192.168.1.57 -p 6379 CLUSTER MEET
192.168.1.58 6381
```



```
(error) ERR Invalid node address specified: 192.168.1.58:6381
```

8. 接下来，进行数据槽（slot）的分配。我们可以在一台主机上使用 `redis-cli`，通过指定主机和端口号的方式进行：

```
redis@192.168.1.57:~> for i in {0..5400}; do redis-cli -h 192.168.1.57 -p 6379
    CLUSTER ADDSLOTS $i; done
OK
...
OK
redis@192.168.1.57:~> for i in {5401..11000}; do redis-cli -h 192.168.1.59 -p
    6380 CLUSTER ADDSLOTS $i; done
OK
...
OK
redis@192.168.1.57:~> for i in {11001..16383}; do redis-cli -h 192.168.1.58 -p
    6381 CLUSTER ADDSLOTS $i; done
OK
...
OK
```

注意

如果我们分配了一个已经被分配过的槽，那么会碰到以下的错误：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.58 -p 6381 CLUSTER ADDSLOTS
11111
```

```
(error) ERR Slot 11111 is already busy
```

如果我们指定的槽的 ID 超出了 0-16383 的范围，则会碰到以下的错误：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.58 -p 6381 CLUSTER ADDSLOTS
22222
```

```
(error) ERR Invalid or out of range slot
```

9. 到这里，我们已经将所有的节点添加到了一个集群中，并分配了所有的 16384 个哈希槽。我们可以通过向集群中的任意一个节点发送 `CLUSTER NODES` 命令来列出所有的节点：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 CLUSTER NODES
eeeabcbab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381@16381 master - 0
    1509885956764 0 connected 11001-16383
549b5b261c765a97b74a374fec49f2ccf30f2acd 192.168.1.58:6380@16380 master - 0
    1509885957000 3 connected
```

```
58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379@16379 myself,master
- 0 1509885955000 2 connected 0-5400
2ff47eb511f0d251eff1d5621e9285191a83ce9f 192.168.1.59:6380@16380 master - 0
1509885957767 1 connected 5401-11000
bc7b4a0c4596759058291f1b8f8de10966b5ald1 192.168.1.59:6379@16379 master - 0
1509885957000 4 connected
7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 192.168.1.57:6381@16381 master - 0
1509885957066 0 connected
```

10. 要实现数据复制，需要将一个节点设置成另一个主节点的从节点。因为我们希望在这个集群中有三个主节点，所以选择三个节点作为从节点：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.59 -p 6379 CLUSTER REPLICATE
58285fa03c19f6e6f633fb5c58c6a314bf25503f
OK
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.58 -p 6380 CLUSTER REPLICATE
2ff47eb511f0d251eff1d5621e9285191a83ce9f
OK
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 CLUSTER REPLICATE
eeeabcb810d500db1d190c592fecbe89036f24f
OK
```

11. 再次使用 CLUSTER NODES 命令查看主从复制的状态，然后使用命令 CLUSTER INFO 获取有关集群的更多信息：

```
192.168.1.57:6379> CLUSTER NODES
eeeabcb810d500db1d190c592fecbe89036f24f 192.168.1.58:6381@16381 master - 0
1510536168000 0 connected 11001-16383
549b5b261c765a97b74a374fec49f2ccf30f2acd 192.168.1.58:6380@16380 slave 2
ff47eb511f0d251eff1d5621e9285191a83ce9f 0 1510536170545 3 connected
58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379@16379 myself,master
- 0 1510536168000 2 connected 0-5400
2ff47eb511f0d251eff1d5621e9285191a83ce9f 192.168.1.59:6380@16380 master - 0
1510536169541 1 connected 5401-11000
bc7b4a0c4596759058291f1b8f8de10966b5ald1 192.168.1.59:6379@16379 slave 58285
fa03c19f6e6f633fb5c58c6a314bf25503f 0 1510536167000 4 connected
7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 192.168.1.57:6381@16381 slave
eeeabcb810d500db1d190c592fecbe89036f24f 0 1510536169000 5 connected
192.168.1.57:6379> CLUSTER INFO
```

```
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
...
cluster_stats_messages_meet_received:2
cluster_stats_messages_received:1483481
```

12. 这样，我们就成功地配置了一个 Redis 集群。我们可以通过设置和获取一个简单的字符串类型的键值对来进行测试：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 -c
192.168.1.57:6379> set foo bar
-> Redirected to slot [12182] located at 192.168.1.58:6381
OK
192.168.1.58:6381> get foo
"bar"
```

7.5.3 工作原理

在前面的示例中，我们一步一步地引导读者学习了配置 Redis 集群的过程。第一步是为每个 Redis 实例准备配置文件：

```
cluster-enabled yes
cluster-config-file nodes-6381.conf
cluster-node-timeout 10000
```

在分别为每个实例指定了不同的监听端口和数据目录之后，我们通过将 `cluster-enabled` 选项设置为 `yes` 来启用集群功能。此外，对于每个 Redis 实例，在 Redis 集群的配置期间都会生成一个集群节点配置文件，并且每当某些集群信息应该被持久化的时候就会被修改。`cluster-config-file` 选项用于设置此配置文件的名字。

不要手动地修改集群节点配置文件。

简单地说，节点超时的含义是：如果经过了指定的超时时间节点无响应，Redis 集群将会触发故障迁移将一个从实例提升为主实例。在下一个案例中，我们将详细讨论这个选项是如何影响集群行为的。

在检查了每个实例的配置文件并执行了一些必要的清理后，我们启动了所有的 Redis 实例。

为了检查某个节点是否正运行在集群模式下，我们可以使用 `redis-cli` 执行 `CLUSTER INFO` 命令，或者在 Redis 实例的日志中搜索“Running mode”。集群节点 ID，即一个 Redis 实例在一个 Redis 集群中的标识，会被记录在日志中。我们还可以通过使用操作系统的 `ps` 命令来判断一个 Redis 实例是否正运行在集群模式下。

在配置并启动了所有的集群节点后，我们接着开始创建一个 Redis 集群。Redis 集群中的节点使用 Redis 集群协议以网状网络拓扑的形式相互通信。因此，我们做的第一步是让每个节点发现彼此，以让它们能在一个集群中正确地工作。为了达到这个目的，我们使用了 `CLUSTER MEET` 命令。

尽管 Redis 集群中的所有节点都需要知道对方的存在，但我们并不需要将 `CLUSTER MEET` 命令发送给每个节点。这是因为，一个节点在发现另一个节点时会向其传播它已知的所有节点的信息（这就是 Redis 文档中所提到的心跳包中 `exchange-of-gossip` 信息的含义）。为了避免混淆，我们可以让一个节点发现所有其他的节点。这样，集群中的所有节点就可以相互通信了。

在 Redis 集群中，数据被按照以下的算法分布到 16384 个哈希槽中：

```
HASH_SLOT = CRC16(key) mod 16384
```

每个主实例都分配了哈希槽的一段范围以存储整个数据集的一部分。因此，我们做的第二步是使用 `CLUSTER ADDSLOTS` 命令在主实例之间分配槽。在完成槽的分配后，我们通过 `CLUSTER NODES` 命令检查集群的状态。该命令的输出如下所示：

```
eeeabcbab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381@16381 master - 0
1510536168000 0 connected 11001-16383
```

每行的格式为：

```
[Node-ID] [Instance-IP:Client-Port@Cluster-Bus-Port] [Master\Slave\Myself] [-\
master's Node-ID if it's a slave] [Ping-Sent timestamp] [Pong-Recv
timestamp] [Config-epoch] [Connection status] [Slots allocated]
```

为了提供数据的冗余，我们通过向欲设为从实例的节点发送 `CLUSTER REPLICATE node-id` 命令来为每个主实例分别分配一个从实例。在配置完主从复制后，我们可以通过 `CLUSTER NODES` 命令的输出来确认主从关系。对于本例而言，我们可以很容易地发现运行在 192.168.1.59:6379 上 ID 为 `bc7b4a0c4596759058291f1b8f8de10966b5a1d1` 的实例是运行在 192.168.1.57:6379 上 ID 为 `58285fa03c19f6e6f633fb5c58c6a314bf25503f` 实例的从实例。

这样，我们就成功地创建了一个 Redis 集群。我们可以通过 `CLUSTER INFO` 命令获得整个集群的状态和指标。

要测试集群工作是否正常，我们使用 `redis-cli` 和 `-c` 选项（用于指定集群模式）连接到集群。出于测试的目的，我们设置并获取了一个简单的字符串键值。我们连接到的节点能够将 `redis-cli`

工具所发送的操作命令重定向到集群中的正确节点。

7.5.4 更多细节

读者可能会觉得创建一个 Redis 集群需要的步骤过多。使用与 Redis 源码一起发布的 `redis-trib.` 脚本来执行 Redis 集群的创建和管理则简单得多。读者可以参阅 Redis 集群的教程来了解更多细节。

对于本示例，我们可以使用以下命令检查集群的状态：

```
redis@192.168.1.57:~> ./script/redis-trib.rb check 192.168.1.57:6379
>>> Performing Cluster Check (using node 192.168.1.57:6379)
M: 58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379
slots:0-5400 (5401 slots) master
...
slots: (0 slots) slave
replicates 58285fa03c19f6e6f633fb5c58c6a314bf25503f
S: 7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 192.168.1.57:6381
slots: (0 slots) slave
replicates eeeabcab810d500db1d190c592fecbe89036f24f
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

注意

对于快速的概念验证测试，我们可以使用 Redis 源码包 `redis/utils/create-cluster` 中的 `create-cluster` 脚本在一台主机上创建一个由 6 个节点（其中有 3 个主实例和 3 个主实例）组成的 Redis 集群。

7.5.5 相关内容

- 官方关于如何创建 Redis 集群的文档，请参阅：<https://redis.io/topics/cluster-tutorial>。
- `CLUSTER NODES` 命令的详细信息，请参阅：<https://redis.io/commands/cluster-nodes>。
- `CLUSTER INFO` 命令的详细信息，请参阅：<https://redis.io/commands/cluster-info>。

7.6 测试 Redis Cluster

在配置了一个 Redis 集群之后，我们需要模拟各种类型的失败来了解集群在意外停机或计划维护情况下的行为。在本案例中，我们将通过一些失败场景来测试上一个案例中搭建的 Redis 集群。之后，我们还将讨论故障迁移的细节。

7.6.1 准备工作

我们需要完成本章配置 *Redis Cluster* 的案例；此外，我们还需要一个安装了 `redis-cli` 的主机作为集群的 Redis 客户端。作为对上一个案例的总结，表 7.2 列出了集群的信息。

表 7.2 集群的信息

实例	IP 地址	端口号	ID	Slots
I_A	192.168.1.57	6379	58285fa03c19f6e6f633fb5c58c6a314bf25503f	0-5400
I_A1	192.168.1.59	6379	bc7b4a0c4596759058291f1b8f8de10966b5a1d1	--
I_B	192.168.1.59	6380	2ff47eb511f0d251eff1d5621e9285191a83ce9f	5401-11000
I_B1	192.168.1.58	6380	549b5b261c765a97b74a374fec49f2ccf30f2acd	--
I_C	192.168.1.58	6381	eeeabcbab810d500db1d190c592fecbe89036f24f	11001-16383
I_C1	192.168.1.57	6381	7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a	--

此外，出于测试的目的，我们将引入由 Redis 的作者 Antirez 提供的测试套件。我们可以按照以下的步骤进行安装：

1. 安装 Ruby 的 Redis 模块：

```
~$ gem install redis
~$ su - redis
~$ cd coding/
```

2. 下载测试套件：

```
~$ mkdir coding
~$ cd coding
~/coding$ git clone https://github.com/antirez/redis-rb-cluster.git
~/coding$ cd redis-rb-cluster/
```


7.6.2 操作步骤

为了测试上一个的案例中配置的 Redis 集群，我们首先通过以下命令启动作为 Redis 客户端的测试程序：

```
~$ ruby coding/redis-rb-cluster/consistency-test.rb 192.168.1.59 6380
1441 R (0 err) | 1441 W (0 err) |
4104 R (0 err) | 4104 W (0 err) |
25727 R (0 err) | 25727 W (0 err) |
...
```

1. 模拟主实例下线：

（1）使用 Redis 的 DEBUG SEGFAULT 命令使实例 I_A 崩溃：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 -c DEBUG SEGFAULT
Error: Server closed the connection
```

（2）检查实例 I_A 的日志：

```
96013:M 15 Nov 14:49:40.224 # Redis 4.0.1 crashed by signal: 11
```

（3）检查测试程序的输出：

```
190927 R (0 err) | 190927 W (0 err) |
201012 R (0 err) | 201012 W (0 err) |
Reading: Connection lost (ECONNRESET)
Writing: Too many Cluster redirections? (last error: MOVED 183
192.168.1.57:6379)
235022 R (2 err) | 235022 W (2 err) |
Reading: Too many Cluster redirections? (last error: MOVED 994
192.168.1.57:6379)
Writing: Too many Cluster redirections? (last error: MOVED 994
192.168.1.57:6379)
....
261178 R (1310 err) | 261179 W (1309 err) |
```

（4）检查实例 I_A1 的日志：

```

35623:S 15 Nov 14:49:40.355 # Connection with master lost.
35623:S 15 Nov 14:49:40.356 * Caching the disconnected master state.
35623:S 15 Nov 14:49:40.410 * Connecting to MASTER 192.168.1.57:6379
35623:S 15 Nov 14:49:40.410 * MASTER <-> SLAVE sync started
35623:S 15 Nov 14:49:40.410 # Error condition on socket for SYNC: Connection
    refused
...
35623:S 15 Nov 14:49:50.452 * Connecting to MASTER 192.168.1.57:6379
35623:S 15 Nov 14:49:50.452 * MASTER <-> SLAVE sync started
35623:S 15 Nov 14:49:50.452 # Error condition on socket for SYNC: Connection
    refused
35623:S 15 Nov 14:49:50.970 * FAIL message received from 2
    ff47eb511f0d251eff1d5621e9285191a83ce9f about 58285
    fa03c19f6e6f633fb5c58c6a314bf25503f
35623:S 15 Nov 14:49:50.970 # Cluster state changed: fail
35623:S 15 Nov 14:49:51.053 # Start of election delayed for 888 milliseconds (
    rank #0, offset 5084834).
35623:S 15 Nov 14:49:51.455 * Connecting to MASTER 192.168.1.57:6379
35623:S 15 Nov 14:49:51.455 * MASTER <-> SLAVE sync started
35623:S 15 Nov 14:49:51.455 # Error condition on socket for SYNC: Connection
    refused
35623:S 15 Nov 14:49:51.957 # Starting a failover election for epoch 8.
35623:S 15 Nov 14:49:51.959 # Failover election won: I'm the new master.
35623:S 15 Nov 14:49:51.959 # configEpoch set to 8 after successful failover
35623:M 15 Nov 14:49:51.959 # Setting secondary replication ID to 744
    a9fb2c14c245888b8e91edd212ae533dd33e3, valid up to offset: 5084835. New
    replication ID is b8fc14c9af26e00c40e964e8c70a8b6001602be1
35623:M 15 Nov 14:49:51.959 * Discarding previously cached master state.
35623:M 15 Nov 14:49:51.960 # Cluster state changed: ok

```

（5）检查所有其他实例的日志：

```
=====I_B=====
```

```

35634:M 15 Nov 14:49:50.969 * Marking node 58285
    fa03c19f6e6f633fb5c58c6a314bf25503f as failing (quorum reached).
35634:M 15 Nov 14:49:50.969 # Cluster state changed: fail
35634:M 15 Nov 14:49:51.959 # Failover auth granted to
    bc7b4a0c4596759058291f1b8f8de10966b5a1d1 for epoch 8

```

```
35634:M 15 Nov 14:49:51.999 # Cluster state changed: ok
=====I_C=====
41354:M 15 Nov 14:50:49.154 * Marking node 58285
fa03c19f6e6f633fb5c58c6a314bf25503f as failing (quorum reached).
41354:M 15 Nov 14:50:49.154 # Cluster state changed: fail
41354:M 15 Nov 14:50:50.143 # Failover auth granted to
bc7b4a0c4596759058291f1b8f8de10966b5a1d1 for epoch 8
41354:M 15 Nov 14:50:50.145 # Cluster state changed: ok
=====I_B1=====
41646:S 15 Nov 14:50:49.154 * FAIL message received from 2
ff47eb511f0d251eff1d5621e9285191a83ce9f about 58285
fa03c19f6e6f633fb5c58c6a314bf25503f
41646:S 15 Nov 14:50:49.154 # Cluster state changed: fail
41646:S 15 Nov 14:50:50.146 # Cluster state changed: ok
=====I_C1=====
27576:S 15 Nov 14:49:50.968 * FAIL message received from 2
ff47eb511f0d251eff1d5621e9285191a83ce9f about 58285
fa03c19f6e6f633fb5c58c6a314bf25503f
27576:S 15 Nov 14:49:50.968 # Cluster state changed: fail
27576:S 15 Nov 14:49:51.959 # Cluster state changed: ok
```

（6）获取集群的当前状态：

```
redis@192.168.1.57:~> ./script/redis-trib.rb check 192.168.1.57:6381
>>> Performing Cluster Check (using node 192.168.1.57:6381)
...
M: bc7b4a0c4596759058291f1b8f8de10966b5a1d1 192.168.1.59:6379
slots:0-5400 (5401 slots) master
0 additional replica(s)
...
[OK] All 16384 slots covered.
```

2. 恢复崩溃的实例。

（1）重启实例 I_A:

```
redis@192.168.1.57:~> bin/redis-server conf/redis-6379.conf
```

（2）检查实例 I_A1 的日志：


```

35623:M 15 Nov 15:00:40.610 * Clear FAIL state for node 58285
fa03c19f6e6f633fb5c58c6a314bf25503f: master without slots is reachable
again.
35623:M 15 Nov 15:00:41.552 * Slave 192.168.1.57:6379 asks for synchronization
35623:M 15 Nov 15:00:41.552 * Partial resynchronization not accepted:
Replication ID mismatch (Slave asked for '9
d2a374586d38080595d4ced9720eeef1c72e1d7', my replication IDs are '
b8fc14c9af26e00c40e964e8c70a8b6001602be1' and '744
a9fb2c14c245888b8e91edd212ae533dd33e3')
35623:M 15 Nov 15:00:41.553 * Starting BGSAVE for SYNC with target: disk
35623:M 15 Nov 15:00:41.553 * Background saving started by pid 113122
113122:C 15 Nov 15:00:41.572 * DB saved on disk
113122:C 15 Nov 15:00:41.572 * RDB: 6 MB of memory used by copy-on-write
35623:M 15 Nov 15:00:41.611 * Background saving terminated with success
35623:M 15 Nov 15:00:41.614 * Synchronization with slave 192.168.1.57:6379
succeeded

```

（3）检查集群的状态：

```

redis@192.168.1.57:~/script> ./redis-trib.rb check 192.168.1.57:6381
>>> Performing Cluster Check (using node 192.168.1.57:6381)
S: 7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 192.168.1.57:6381
slots: (0 slots) slave
replicates eeeabcbab810d500db1d190c592fecbe89036f24f
M: eeeabcbab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381
slots:11001-16383 (5383 slots) master
1 additional replica(s)
M: 2ff47eb511f0d251eff1d5621e9285191a83ce9f 192.168.1.59:6380
slots:5401-11000 (5600 slots) master
1 additional replica(s)
M: bc7b4a0c4596759058291f1b8f8de10966b5a1d1 192.168.1.59:6379
slots:0-5400 (5401 slots) master
1 additional replica(s)
S: 58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379
slots: (0 slots) slave
replicates bc7b4a0c4596759058291f1b8f8de10966b5a1d1
S: 549b5b261c765a97b74a374fec49f2ccf30f2acd 192.168.1.58:6380
slots: (0 slots) slave

```

```
replicates 2ff47eb511f0d251eff1d5621e9285191a83ce9f
```

```
[OK] All nodes agree about slots configuration.
```

```
>>> Check for open slots...
```

```
>>> Check slots coverage...
```

```
[OK] All 16384 slots covered.
```

3. 模拟从实例下线:

(1) 使用 Redis 的 `DEBUG SEGFAULT` 命令使实例 `I_C1` 崩溃:

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 -c DEBUG SEGFAULT
Error: Server closed the connection
```

(2) 检查实例 `I_C` 的日志:

```
41354:M 15 Nov 15:13:03.564 # Connection with slave 192.168.1.57:6381 lost.
41354:M 15 Nov 15:13:13.750 * FAIL message received from
      bc7b4a0c4596759058291f1b8f8de10966b5a1d1 about 7
      e06908bd0c7c3b23aaa17f84d96ad4c18016b1a
```

(3) 查看实例 `I_A` 的日志:

```
112615:S 15 Nov 15:12:15.528 * FAIL message received from
      bc7b4a0c4596759058291f1b8f8de10966b5a1d1 about 7
      e06908bd0c7c3b23aaa17f84d96ad4c18016b1a
```

(4) 检查集群的状态:

```
redis@192.168.1.57:~> ./script/redis-trib.rb check 192.168.1.57:6379
>>> Performing Cluster Check (using node 192.168.1.57:6379)
...
M: eeeabcab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381
  slots:11001-16383 (5383 slots) master
  0 additional replica(s)
...
[OK] All 16384 slots covered.
```

4. 模拟一个分片的主实例和从实例下线:

(1) 我们之前已经让实例 `I_C1` 崩溃了, 现在我们将实例 `I_C` 下线:

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.58 -p 6381 -c DEBUG SEGFAULT
Error: Server closed the connection
```

(2) 检查实例 I_A1 的日志:

```
35623:M 15 Nov 15:47:29.855 # Cluster state changed: fail
```

(3) 检查集群的状态:

```
redis@192.168.1.57:~> ./script/redis-trib.rb check 192.168.1.57:6379
>>> Performing Cluster Check (using node 192.168.1.57:6379)
S: 58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379
slots: (0 slots) slave
replicates bc7b4a0c4596759058291f1b8f8de10966b5a1d1
M: 2ff47eb511f0d251eff1d5621e9285191a83ce9f 192.168.1.59:6380
slots:5401-11000 (5600 slots) master
1 additional replica(s)
M: bc7b4a0c4596759058291f1b8f8de10966b5a1d1 192.168.1.59:6379
slots:0-5400 (5401 slots) master
1 additional replica(s)
S: 549b5b261c765a97b74a374fec49f2ccf30f2acd 192.168.1.58:6380
slots: (0 slots) slave
replicates 2ff47eb511f0d251eff1d5621e9285191a83ce9f
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[ERR] Not all 16384 slots are covered by nodes.
```

(4) 检查测试程序的输出:

```
~$ ruby coding/redis-rb-cluster/consistency-test.rb 192.168.1.57 6379
0 R (6261 err) | 0 W (6261 err) |
...
Reading: CLUSTERDOWN The cluster is down
Writing: CLUSTERDOWN The cluster is down
0 R (7727 err) | 0 W (7727 err) |
```


7.6.3 工作原理

通过运行测试脚本 `consistency-test.rb`，我们可以定期读写一个 Redis 集群。在脚本运行期间，写入、读取以及错误的数量都会被记录下来。任何的数据不一致都会被捕获到。

我们进行的第一个测试是将一个主实例下线。我们使用 `DEBUG SEGFAULT` 命令使实例 ID 为 `58285fa03c19f6e6f633fb5c58c6a314bf25503f` 的实例 `I_A` 崩溃，该命令会让 Redis 实例发生段错误而退出，如实例 `I_A` 的日志所示。从测试程序的输出中我们可以清楚地看到，某些写入和读取请求失败了一段时间。稍后，集群恢复了对请求的处理。实际上，集群不能工作的这段时间与 `I_A1` 的故障迁移过程有关。让我们深入 `I_A1` 的日志，来了解更多关于故障迁移的信息。

首先，从实例在 `14:49:40` 发现与主实例的连接断开，然后在之后的大约 10 秒钟（选项 `cluster-timeout` 指定的时间）内试图重新连接到主实例。稍后，在 `14:49:50`，通过收到 `FAIL` 消息，确认 `I_A` 节点已经失效，且集群的状态也被认为失效了。`I_A1` 得到了两个存活主实例的投票，随后在 `14:49:51` 成为了新的主实例。最后，集群的状态再次被更改为 `OK`。从其他主实例和从实例的日志所显示的时间轴中我们可以看到，`I_B`（实例 ID 是 `2ff47eb511f0d251eff1d5621e9285191a83ce9f`）首先将 `I_A` 标记为 `FAIL` 并把这个消息广播给了所有的从实例。

尽管所有的槽都被覆盖了且集群的状态也是 `OK` 的，但检查集群时我们发现为 `0-5400` 槽服务的新主实例并没有副本。

我们进行的第二个测试是重启在之前测试中崩溃的实例。通过再次启动实例 `I_A`，我们发现它成为了实例 `I_A1`（现在是故障迁移后的主实例）的从实例。我们还发现，`I_A` 的 `FAIL` 状态被移除并启动了重新同步。最后，为 `0-5400` 提供服务的新主实例 `I_A1` 拥有了一个由 `I_A` 提供的副本。

我们进行的第三个测试是让一个从实例崩溃。我们选择的节点是 `I_C1`。在将 `I_C1` 下线后，因为崩溃的实例是一个不为槽提供服务的从实例，所以我们发现集群的状态并没有发生改变。

我们进行的最后一个测试是把一个分片的主实例和从实例都下线了。由于并非所有的槽都能够被覆盖，所以集群的状态变成了 `FAIL`。因此，测试程序报错：`CLUSTERDOWN The cluster is down.`

7.6.4 更多细节

实际上，在 `FAIL` 消息之前，还有一个 `PFAIL`（可能失败，possible failure）的状态会在集群的节点中传播。状态 `PFAIL` 的意思是，一个节点 A（不管是主实例还是从实例）可能在另一个节点 B 不可达时将其标记为 `PFAIL`。然后，这个节点 A 会通过心跳传播这个信息。之后，当 A 在 `NODE_TIMEOUT * 2` 的时间范围内接收到大多数主实例认为 B 处于 `PFAIL` 或 `FAIL` 状态的消息时，A 就会将 `PFAIL` 修改为 `FAIL` 并将消息广播到其他节点。

因此，在此案例中，如果几乎同时让两个主实例崩溃，即使两个主实例分别拥有从实例，也不会进行故障迁移。这是因为，大多数的主实例都下线了。结果，`PFAIL` 都不会被设置为 `FAIL`，故而

也不会发生故障迁移。出于这个原因，作为一个生产实践，我们不应该把大多数主实例部署在同一台主机上。

7.6.5 相关内容

- 有关 Redis 集群的更多细节，请参阅 *Redis 集群规范*：<https://redis.io/topics/cluster-spec>。
- 有关 `DEBUG SEGFAULT` 命令的更多细节，请参阅：<https://redis.io/commands/debug-segfault>。
- 我们还可以深入 Redis 集群的源代码来了解更多信息，请参阅：<http://download.redis.io/redis-stable/src/cluster.c>。
- 一个 Redis 作者 Antirez 讨论 Redis 集群的 PPT，请参阅：https://redis.io/presentation/Redis_Cluster.pdf。

7.7 管理 Redis Cluster

由于全连接的网格拓扑和故障迁移机制，Redis 集群比单一的主从架构要复杂得多。对于那些管理集群的人来说，学习如何获得集群的运行拓扑和状态是非常重要的。此外，使用 Redis 集群的最大好处之一在于其可以轻松地添加或删除节点。当我们使用 Redis 集群时，这些操作都是非常常见的。

在本案例中，我们将学习如何在 Redis 集群中执行常见的运维操作。

7.7.1 准备工作

我们需要完成本章配置 *Redis Cluster* 的案例；此外，我们还需要一个安装了 `redis-cli` 的主机作为集群的 Redis 客户端。对于演示环境，我们可以参考测试 *Redis Cluster* 案例中准备工作一节中提供的表格。

7.7.2 操作步骤

接下来，让我们按照以下的步骤学习如何管理 Redis 集群。

1. 获取集群的状态：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 -c CLUSTER INFO
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
...
```

```
cluster_stats_messages_pong_received:113
cluster_stats_messages_fail_received:2
cluster_stats_messages_auth-req_received:2
cluster_stats_messages_received:233
```

2. 检查集群中节点的状态:

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 -c CLUSTER NODES
eeeeabcab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381@16381 master - 0
1510818967000 0 connected 11001-16383
...
58285fa03c19f6e6f633fb5c58c6a314bf25503f 192.168.1.57:6379@16379 slave
bc7b4a0c4596759058291f1b8f8de10966b5ald1 0 1510818968000 14 connected
2ff47eb511f0d251eff1d5621e9285191a83ce9f 192.168.1.59:6380@16380 master - 0
1510818967067 1 connected 5401-11000
```

3. 触发手动故障迁移，让一个从实例提升为主实例:

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 -c CLUSTER
FAILOVER
OK
```

4. 从指定的主实例中获取从实例的信息:

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6381 -c CLUSTER SLAVES
7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a
1) "eeeeabcab810d500db1d190c592fecbe89036f24f 192.168.1.58:6381@16381 slave 7
e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 0 1510819599257 17 connected"
```

5. 向正在运行的集群添加一个分片（主实例和它的从实例）。

（1）准备主实例和监听 6382 端口的从实例的配置文件:

```
redis@192.168.1.57:~> cat conf/redis-6382.conf
daemonize yes
pidfile "/redis/run/redis-6382.pid"
port 6382
bind 192.168.1.57
logfile "/redis/log/redis-6382.log"
dbfilename "dump-6382.rdb"
dir "/redis/data"
```



```
...
cluster-enabled yes
cluster-config-file nodes-6382.conf
cluster-node-timeout 10000
```

(2) 分别在 192.168.1.57 和 192.168.1.59 上启动这两个实例：

```
~> bin/redis-server conf/redis-6382.conf
```

(3) 添加主实例和从实例：

```
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 -c CLUSTER MEET
192.168.1.57 6382
OK
redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6379 -c CLUSTER MEET
192.168.1.59 6382
OK
```

提示

如果在添加一个节点时发生了如下的错误，则表示该节点存有一些数据或节点配置文件已经存在：[ERR] Node 192.168.145.128:6382 is not empty. Either the node already knows other nodes (check with CLUSTER NODES) or contains some key in database 0。

我们可以采用如下的步骤来重置节点：

- (1) 进入 dir 选项指定的目录并删除节点配置文件。
- (2) 使用 redis-cli 连接到节点并执行命令 FLUSHDB。
- (3) 删除所有的 RDB 和 AOF 文件。
- (4) 列出两个节点的 ID：

```
redis@192.168.1.57:~> script/redis-trib.rb check
192.168.1.57:6381
>>> Performing Cluster Check (using node 192.168.1.57:6381)
M: 7e06908bd0c7c3b23aaa17f84d96ad4c18016b1a 192.168.1.57:6381
slots:11001-16383 (5383 slots) master
1 additional replica(s)
M: a693372f4feeb1b1cf2bd4cb1f4881d2caa0d7a7c 192.168.1.57:6382
slots: (0 slots) master
```

```

0 additional replica(s)
...
M: 7abel13b549b66218990c9fc8e2d209803f03665d 192.168.1.59:6382
slots: (0 slots) master
0 additional replica(s)
[OK] All 16384 slots covered.

```

(5) 配置两个实例的主从复制：

```

redis@192.168.1.57:~> bin/redis-cli -h 192.168.1.57 -p 6382 -c
CLUSTER REPLICATE 7
abel13b549b66218990c9fc8e2d209803f03665d
OK

```

(6) 检查主从复制关系：

```

redis@192.168.1.57:~> script/redis-trib.rb check
192.168.1.57:6381
>>> Performing Cluster Check (using node 192.168.1.57:6381)
...
S: a693372f4feelb1cf2bd4cb1f4881d2caa0d7a7c 192.168.1.57:6382
slots: (0 slots) slave
replicates 7abel13b549b66218990c9fc8e2d209803f03665d
M: 7abel13b549b66218990c9fc8e2d209803f03665d 192.168.1.59:6382
slots: (0 slots) master
1 additional replica(s)
...
[OK] All 16384 slots covered.

```

(7) 将 500 个槽从实例 I_A 迁移到新添加的实例：

```

redis@192.168.1.57:~> script/redis-trib.rb reshard --from
bc7b4a0c4596759058291f1b8f8de10966b5a1d1 --to 7
abel13b549b66218990c9fc8e2d209803f03665d --slots 100 --yes
192.168.1.57:6379
>>> Performing Cluster Check (using node 192.168.1.57:6379)
...
M: 7abel13b549b66218990c9fc8e2d209803f03665d 192.168.1.59:6382
slots: (0 slots) master
0 additional replica(s)

```

...

S: a693372f4feelb1cf2bd4cb1f4881d2caa0d7a7c 192.168.1.57:6382

slots: (0 slots) slave

replicates bc7b4a0c4596759058291f1b8f8de10966b5ald1

...

[OK] All 16384 slots covered.

Ready to move 100 slots.

Source nodes:

M: bc7b4a0c4596759058291f1b8f8de10966b5ald1 192.168.1.59:6379

slots:0-5400 (5401 slots) master

2 additional replica(s)

Destination node:

M: 7abel3b549b66218990c9fc8e2d209803f03665d 192.168.1.59:6382

slots: (0 slots) master

0 additional replica(s)

Resharding plan:

Moving slot 0 from bc7b4a0c4596759058291f1b8f8de10966b5ald1

Moving slot 1 from bc7b4a0c4596759058291f1b8f8de10966b5ald1

Moving slot 2 from bc7b4a0c4596759058291f1b8f8de10966b5ald1

...

Moving slot 0 from 192.168.1.59:6379 to 192.168.1.59:6382: ..

...

Moving slot 98 from 192.168.1.59:6379 to 192.168.1.59:6382:

...

Moving slot 99 from 192.168.1.59:6379 to 192.168.1.59:6382:

(8) 再次检查集群状态:

```
redis@192.168.1.57:~> script/redis-trib.rb check 192.168.1.57:6381
```

```
>>> Performing Cluster Check (using node 192.168.1.57:6381)
```

...

S: a693372f4feelb1cf2bd4cb1f4881d2caa0d7a7c 192.168.1.57:6382

slots: (0 slots) slave

replicates bc7b4a0c4596759058291f1b8f8de10966b5ald1

M: 7abel3b549b66218990c9fc8e2d209803f03665d 192.168.1.59:6382

slots:0-99 (100 slots) master

1 additional replica(s)

...


```
[OK] All 16384 slots covered.
```

6. 从一个运行的集群中删除一个分片（主实例和它的从实例）：

（1）删除从节点：

```
redis@192.168.1.57:~> script/redis-trib.rb del-node 192.168.1.57:6379
a693372f4fee1b1cf2bd4cb1f4881d2caa0d7a7c
>>> Removing node a693372f4fee1b1cf2bd4cb1f4881d2caa0d7a7c from cluster
192.168.1.57:6379
>>> Sending CLUSTER FORGET messages to the cluster...
>>> SHUTDOWN the node.
```

（2）迁移分配给待删除主实例的槽：

```
redis@192.168.1.57:~> script/redis-trib.rb reshard --to
bc7b4a0c4596759058291f1b8f8de10966b5a1d1 --from 7
abel13b549b66218990c9fc8e2d209803f03665d --slots 100 --yes
192.168.1.57:6379
```

（3）删除主节点：

```
redis@192.168.1.57:~> script/redis-trib.rb del-node 192.168.1.57:6379 7
abel13b549b66218990c9fc8e2d209803f03665d
>>> Removing node 7abel13b549b66218990c9fc8e2d209803f03665d from cluster
192.168.1.57:6379
>>> Sending CLUSTER FORGET messages to the cluster...
>>> SHUTDOWN the node.
```

（4）检查集群状态：

```
redis@192.168.1.57:~> script/redis-trib.rb check 192.168.1.57:6381
```

7.7.3 工作原理

上一节中显示的命令非常直白。我们应该知道的一点是，槽在实例间重新分配期间，对集群的写入和读取请求不会受到影响。

由于本书篇幅的限制，我们没有介绍有关槽的命令，例如 `CLUSTER ADDSLOTS`、`CLUSTER DELSLOTS` 或 `CLUSTER SETSLOT`。实际上，了解这些命令的工作原理非常有助于理解集群是如何在不影响请求处理的情况下执行槽的重新分片操作的。

7.7.4 更多细节

在第9章管理 *Redis* 中数据迁移的案例中,我们将展示如何将数据从单个 *Redis* 实例迁移到 *Redis* 集群。

7.7.5 相关内容

- 有关 *Redis* 集群中的命令,请参阅官方文档: <https://redis.io/commands#cluster>。
- 使用 `help` 选项可以了解更多有关 `redis-trib.rb` 脚本的细节。

第 8 章

生产环境部署

在本章中，我们将学习下列案例：

- 在 Linux 上部署 Redis。
- Redis 安全相关设置。
- 配置客户端连接选项。
- 配置内存策略。
- 基准测试。
- 日志。

8.1 本章概要

正如我们在第 1 章开始使用 *Redis* 中所展示的，配置一个测试用途的 *Redis* 实例是很便捷的。而将 *Redis* 服务器部署到生产环境，则需要考虑更多的因素。在本章中，我们将重点讨论如何将 *Redis* 部署到 Linux 系统中。我们将从操作系统级的优化开始，配置一个能够用于实际生产环境的 *Redis* 服务器。然后，我们将讨论服务器端有关客户端连接的参数，以及如何在生产环境中安全加固 *Redis*。我们还会讨论如何配置内存策略。最后，我们将介绍 *Redis* 的日志选项和基准测试工具。

8.2 在 Linux 上部署 Redis

虽然我们可以通过编译源代码的方式在几乎所有的现代操作系统上安装 *Redis*，但运行 *Redis*，Linux 是最常见的操作系统。在启动 *Redis* 实例之前，通常需要将一些 Linux 内核和操作系统级的参

数设置为恰当的值,以便在生产环境中发挥最高性能。在本案例中,我们将介绍一些关键的内核和操作系统参数或配置。

8.2.1 准备工作

我们需要按照第1章开始使用 Redis 中下载和安装停止 Redis 的案例所描述的步骤安装一个 Redis 服务器。

8.2.2 操作步骤

接下来,让我们按照以下的步骤学习如何在 Linux 上部署 Redis。

1. 设置下列与内存相关的内核参数:

```
~$ sudo sysctl -w vm.overcommit_memory=1
~$ sudo sysctl -w vm.swappiness=0
```

使用如下的命令来持久化地保存这些参数:

```
echo vm.overcommit_memory=1" >> /etc/sysctl.conf
echo "vm.swappiness=0" >> /etc
/sysctl.conf
```

使用如下的命令来检查这些参数是否已被设置:

```
~$ sudo sysctl vm.overcommit_memory vm.swappiness
vm.overcommit_memory = 1
vm.swappiness = 0
```

2. 此外,禁用透明大页(transparent huge page)功能:

```
~$ sudo su -
~# echo never > /sys/kernel/mm/transparent_hugepage/enabled
~$ echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

使用如下的命令来持久化地保存这些设置:

```
~ # cat >> /etc/rc.local << EOF
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
EOF
```

注意

对于 RedHat Linux, 我们可以使用 `echo never > /sys/kernel/mm/redhat_transparent_hugepage/enabled` 修改 `/etc/rc.local`。

使用如下的命令来检查这些参数是否已被设置：

```
~$ cat /sys/kernel/mm/transparent_hugepage/enabled
always madvise [never]
~$ cat /sys/kernel/mm/transparent_hugepage/defrag
always madvise [never]
```

3. 对于网络的优化, 我们设置下列与网络相关的内核参数：

```
~$ sudo sysctl -w net.core.somaxconn=65535
~$ sudo sysctl -w net.ipv4.tcp_max_syn_backlog=65535
```

使用如下的命令来持久化地保存这些参数：

```
echo "net.core.somaxconn=65535" >> /etc/sysctl.conf
echo "net.ipv4.tcp_max_syn_backlog=65535" >> /etc/sysctl.conf
```

使用如下的命令来检查这些参数是否已被设置：

```
~$ sudo sysctl net.core.somaxconn net.ipv4.tcp_max_syn_backlog
net.core.somaxconn = 65535
net.ipv4.tcp_max_syn_backlog = 65535
```

4. 要将进程能够打开的文件数设为更高的值, 我们需要先切换到启动 Redis 进程的用户, 然后执行 `ulimit` 命令：

```
~$ su - redis~$ ulimit -n 288000
```

注意

我们必须将 `nofile` 设为一个小于 `/proc/sys/fs/file-max` 的值。因此, 在设置之前, 我们需要使用 `cat` 命令检查 `/proc/sys/fs/file-max` 的值。

要持久化地保存这个参数, 可以将如下两行添加到 `/etc/security/limits.conf` 中：

```
redis soft nofile 288000
redis hard nofile 288000
```

使用如下的命令来检查这些参数是否已被设置：

```
~$ ulimit -Hn -Sn
open files                (-n) 288000
open files                (-n) 288000
```

8.2.3 工作原理

下面，让我们来讨论上一节中提到的每个配置或参数。我们调整的第一个设置是 `overcommit_memory`。在第 6 章持久化中曾提到的，Redis 在后台持久化时利用了写时复制（**Copy-On-Write**, **COW**）的优点。这意味着在 Redis 中不需要使用与数据集大小相同的空闲内存。但是，Linux 在默认情况下可能会检查是否有足够的空闲内存来复制父进程的所有内存页；而这可能会导致进程由于 OOM（Out of Memory）而崩溃。如果出现这种情况，我们会在 Redis 的运行日志中发现如下的错误：

```
[1524] 24 Sep 10:00:56.037 # Can't save in background: fork: Cannot allocate
memory
```

实际上，在我们碰到这个问题时，如果仔细地查看 Redis 的运行日志，会发现如下的内容：

```
5885:M 19 Nov 09:18:29.324 # WARNING overcommit_memory is set to 0! Background
save may fail under low memory condition. To fix this issue add 'vm.
overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the
command 'sysctl vm.overcommit_memory=1' for this to take effect.
```

要解决这个问题，我们需要将 `overcommit_memory` 设为 1，表示当一个程序调用诸如 `malloc()` 等函数分配内存时，即使系统知道没有足够的内存空间函数也会执行成功。第二个内存相关的配置是 `vm.swappiness`，该参数定义了 Linux 内核将内存中的内容拷贝到交换分区的大小（及频率）。

提示

`swappiness` 参数的值越大，内核在进行内存交换时就会越激进。

启用内存交换（译者注：也译作虚拟内存）后，Redis 可能会尝试访问位于磁盘中的内存页。这将导致 Redis 进程被磁盘 I/O 操作（可能是一个缓慢的过程）阻塞。我们通常需要充分发挥 Redis 高速的处理能力，因此被交换分区拖慢 Redis 的速度是不可取的。故而，我们总是建议将 `vm.swappiness` 设为 0。值得一提的是，将这个参数设置为 0 对于不同的内核版本来说含义是不同的：

- 对于 Linux 3.5 和更新版本而言，`swappiness` 为 0 表示完全禁用交换分区。
- 对于 Linux 3.4 和更老版本而言，`swappiness` 为 0 表示只在避免时使用交换分区。

对于 Redis 来说，我们首先应尽可能得避免使用 `swappiness`。此外，在 Redis 实例没有足够的内存可以使用时，我们宁愿让 Redis 进程被杀死，也不希望它被交换分区拖慢。如果 Redis 服务能够做到快速失败（fail-fast），那么 HA 或集群机制就能够妥善处理崩溃的情况。因此，无论 Linux 内核是什么版本，我们总是建议将这个参数设为 0。

我们接下来进行的调整是禁用 Linux 内核提供的透明大页功能。这一功能可能会导致持久化时的子进程创建缓慢。因此，我们建议禁用该功能；否则，在实例启动时，在 Redis 的运行日志中会收到一个警告信息：

```
3248:M 21 Oct 22:16:23.485 # WARNING you have Transparent Huge Pages (THP)
support enabled in your kernel. This will create latency and memory usage
issues with Redis. To fix this issue run the command 'echo never > /sys/
kernel/mm/transparent_hugepage/enabled' as root, and add it to your /etc/
rc.local in order to retain the setting after a reboot. Redis must be
restarted after THP is disabled.
```

对于网络而言，我们将 `net.core.somaxconn` 和 `net.ipv4.tcp_max_syn_backlog` 设为了比默认值 128 大很多的 65535。前一个内核参数设置了传递给 `listen` 函数的 `backlog` 参数的上限，后一个参数设置了挂起连接的最大队列长度。在 Redis 中，有一个默认值为 511 的选项 `tcp-backlog`；将这些值设置得大一点可以优化 TCP 连接。如果读者不把 `net.core.somaxconn` 设为高于 511 的值，那么当实例启动时会在 Redis 运行日志中收到如下的警告信息：

```
WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/
net/core/somaxconn is set to the lower value of 128.
```

最后，我们还设置了一个进程能够打开的最大文件数。我们需要确保这个操作系统参数的值高于 Redis 中的 `maxclients` 选项。如果这个参数不满足上述的条件，那么我们会在 Redis 运行日志中看到如下的日志：

```
# You requested maxclients of 10000 requiring at least 10032 max file
descriptors.
# Redis can't set maximum open files to 10032 because of OS error: Operation
not permitted
# Current maximum open files is 4096. maxclients has been reduced to 4064 to
compensate for low ulimit. If you need higher maxclients increase 'ulimit
-n'.
```

8.2.4 更多细节

在本章配置客户端连接选项的案例中，我们将展示如何设置 `maxclients` 和 `tcp-backlog` 选项。

8.2.5 相关内容

- 读者可以在 Redis 的 FAQ 中找到更多的细节：<https://redis.io/topics/faq>。
- 有关 `swappiness` 参数的更多细节，请参阅维基百科上的相关内容：<https://en.wikipedia.org/wiki/Swappiness>。
- 有关透明大页的更多细节，请参阅相关文档：<https://www.kernel.org/doc/Documentation/vm/transhuge.txt>。
- 有关 Redis 部署的更多细节，请参阅相关文档：<https://redis.io/topics/admin>。

8.3 Redis 安全相关设置

安全显然是任何生产环境中最基本的问题之一。然而，因为 Redis 的设计思想是它将部署在所有客户端都可信的环境中，所以其本身在安全性方面只提供了非常有限的功能。Redis 最初的设计思想更侧重于优化极限性能和简洁性，而没有考虑支持完整的身份验证和访问控制。虽然 Redis 的安全大部分依赖于 Redis 之外（操作系统、防火墙）的机制，但我们仍然可以做一些工作来保护 Redis 服务器以免受非法的访问和攻击。在本案例中，我们将讨论在生产环境中加固 Redis 的一些常见做法。

8.3.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

8.3.2 操作步骤

接下来，让我们学习如何通过恰当的网络配置来加固 Redis。因为公网上任何不受信的客户端都可以访问，所以将 Redis 生产服务器暴露在公网上是不明智的。因此，我们必须配置 Redis 绑定到属于受信网络中的 IP 地址上，具体步骤如下所示：

1. 修改配置文件中的 `bind` 和 `port` 来更新绑定的 IP 地址和端口：

```
bind 127.0.0.1 192.168.0.31
port 36379
```

2. 不要忘记更新主从复制和 Sentinel 配置中的相应参数：

```
#Replica's configuration (redis.conf)
slaveof 192.168.0.31 36379

#Sentinel's configuration (sentinel.conf)
sentinel monitor mymaster 192.168.0.31 36379
```

此外，我们还可以让 Redis 不监听任何网络接口，而监听 Unix 域套接字。这样，只有拥有套接字访问权限的本地客户端才可以访问 Redis。

1. 我们可以在配置文件中设置 `unixsocket` 和 `unixsocketperm` 参数，然后将 `port` 设为 0，使 Redis 只监听 Unix 套接字：

```
port 0
unixsocket /var/run/redis/redis.sock
unixsocketperm 766
```

2. 我们可以使用 `redis-cli -s <socket>` 连接到监听在 Unix 套接字上的 Redis 服务器：

```
~$ bin/redis-cli -s /var/run/redis/redis.sock
redis /var/run/redis/redis.sock>
```

通过密码验证来保护 Redis 通常是一个良好的实践。Redis 提供了一个非常简单的验证机制来防止未经授权的访问。我们可以给整个 Redis 服务器设置一个密码，让所有客户端在通过密码进行身份验证后才能执行命令。

1. 要设置 Redis 服务器的密码，只需在配置文件的 `requirepass` 部分中添加明文密码即可：

```
requirepass foobared
```

2. 要访问一个启用了身份验证的 Redis 服务器，需要使用 `AUTH` 命令：

```
$ bin/redis-cli
127.0.0.1:6379> SET test 123
(error) NOAUTH Authentication required.
127.0.0.1:6379> AUTH foobared
OK
127.0.0.1:6379> SET test 123
OK
```


3. 如果 Redis 服务器启用了主从复制，则需要将主实例的密码添加到所有从实例的配置文件中：

```
masterauth foobared
```

Redis 还支持将某些命令重命名。如果我们将某些诸如 CONFIG 或 FLUSHDB 的危险命令重命名为难以猜测的名称，那么客户端将几乎不可能执行这些命令。如果需要更高的安全性，我们也可以将这些命令完全禁用。

1. 要重命名命令，在 Redis 的配置文件中添加 `rename-command` 参数即可：

```
rename-command CONFIG F3E9DD63CDDAD0EBBA50EC22D78A00F34F6B9CB1
```

2. 要完全禁用某个命令，只需要将其重命名为空字符串即可：

```
rename-command CONFIG ""
```

8.3.3 工作原理

为了增强 Redis 的安全性，我们介绍了三个配置选项。如前所述，我们强烈建议读者不要向公网暴露 Redis。如果 Redis 服务器不得不通过公网访问，那么使用一个不同于默认端口 6379 的端口号能够略微降低端口扫描器带来的风险（尽管通过 Redis 通信协议来检测端口背后的服务器并不困难）。

Redis 的身份验证密码以明文保存在配置文件中。因此，我们应该严格设置 Redis 配置文件的访问权限，以防止非管理员访问。此外，因为服务器和客户端之间的通信是未加密的，所以 AUTH 命令中的密码可能会被嗅探器嗅探到，故而 Redis 的密码机制应该被认为是最后一道防线。

Redis 主实例中的密码选项不会被从实例继承，因此我们必须分别为从实例设置密码。即使服务器已被攻破的情况下，重命名或禁用危险命令也可以在一定程度上阻止高风险的操作。但是，由于重命名的配置也保存在 Redis 的配置文件中，所以必须使用严格的访问权限来保护配置文件。

8.3.4 更多细节

Redis 有一个特殊的模式，称为保护模式（protected mode），可以通过在配置文件中设置 `protected-mode yes` 来启用。如果 Redis 服务器启用了保护模式，且同时被配置为监听所有接口（在配置文件中没有 `bind` 参数），且禁用了密码验证，那么 Redis 只会响应来自本地回环接口和 Unix 套接字的查询；来自其他接口的查询会被拒绝。

保护模式在默认情况下是启用的，除非 Redis 必须监听所有接口且网络环境是安全的，否则即使禁用了密码验证也不应禁用保护模式。在操作系统中，我们可以采取其他的预防措施来保护 Redis。

一种措施是，不使用 root 权限的用户运行 Redis 而使用有限权限的专门用户来运行 Redis。同时，这个用户应该在 SSH 登录中禁用。另一种措施是，在操作系统的防火墙中设置白名单，只允许受信任的 IP 地址访问 Redis。

8.3.5 相关内容

- 有关 Redis 安全的官方主题，请参阅：<https://redis.io/topics/security>。
- 有关 Redis 安全的几个要点，请参阅：<http://antirez.com/news/96>。

8.4 配置客户端连接选项

对于 Redis 的客户端来说，服务器端有几个重要的配置参数。在本案例中，我们将介绍这些参数，并学习这些客户端连接选项的最佳实践。

8.4.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

8.4.2 操作步骤

接下来，让我们按照以下的步骤学习如何配置客户端连接选项。

1. 如果要配置客户端的网络参数，将以下内容添加到 Redis 配置文件 `redis.conf` 中即可：

```
timeout 0
tcp-backlog 511
```

2. 如果要配置 Redis 实例的最大客户端数量和 `tcp-keepalive` 时间，将以下内容添加到 Redis 和 Sentinel 的配置文件中即可：

```
maxclients 10000
tcp-keepalive 300
```

3. 如果要配置客户端的缓冲区参数，在 Redis 配置文件 `redis.conf` 中添加以下内容即可：

```
client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 512mb 256mb 60
client-output-buffer-limit pubsub 32mb 8mb 60
```


8.4.3 工作原理

下面，让我们学习上一节中提到的每个参数。我们为 Redis 实例设置的两个参数都与 Redis 和客户端之间的网络连接有关。第一个选项是 `timeout`，表示 Redis 服务器将在客户端空闲 N 秒后关闭连接。实际上，指定的 `timeout` 会被传递给 `setsockopt()` 来设置 `SO_SNDTIMEO` 选项。我们可以参阅套接字的帮助页面了解详情。

第二个选项是 `tcp-backlog`，该选项用于设置挂起套接字请求队列的大小。在在 Linux 上部署 Redis 的案例中，我们介绍了内核参数 `somaxconn`。我们应该确保 `tcp-backlog` 的值小于 `somaxconn`。在真实的生产环境中，大多数情况下，我们建议将其设为一个高于默认值 10000 的值。

`maxclients` 和 `tcp-keepalive` 选项对 Redis 实例和 Sentinel 都有效。我们强烈建议读者确保为 Redis 实例及 Sentinel 服务设置了这两个参数。`maxclients` 选项限制了来自客户端的最大连接数。一旦连接数超过这个限制，新的连接请求将收到一条如下所示的错误消息并被立即拒绝：

```
ERR max number of clients reached.
```

下一个选项是 `tcp-keepalive`。如果将该选项设为非零值，那么服务器将按照指定的时间间隔发送 TCP ACK 来通知网络设备在此期间连接仍然活跃。如果一个客户端没有响应 TCP 活跃的消息，那么服务器将客户端视为下线并将连接关闭。当客户端和 Redis 服务器之间存在硬件防火墙时，这个选项非常有用。

例如，Juniper 防火墙在客户端和它本身之间的连接空闲超过 1800 秒时会将连接切断。但是，客户端和服务端都不会得到连接已断开的通知。因此，从服务器的角度来看，连接仍然是活跃的。如果我们禁用了该选项，服务器就不会释放连接。另一方面，从客户端的角度来看，一旦它发现连接断开就会重新连接到 Redis 服务器。这样，会导致连接数持续增加。因此，为了避免此类连接问题的发生，保持默认的设置更为可取。在默认情况下，该选项的值被设为 300。

最后的三个参数都与客户端输出缓冲区相关。Redis 不会直接将命令的输出发送给客户端，而是首先将结果填充到每个连接的输出缓冲区中，然后再将其内容一次性地发送出去。不同类型客户端（普通连接、pub/sub 连接、从实例连接）的输出缓冲区大小是不同的。该配置参数的格式为：

```
client-output-buffer-limit <class> <hard limit> <soft limit> <soft seconds>
```

对于输出缓冲区大小，既有硬性限制也有软性限制。我们已经在第 5 章复制（Replication）中复制机制的故障诊断的案例中学习了输出缓冲区硬性限制和软性限制的含义。

8.4.4 相关内容

- 读者可以参阅以下链接了解有关 Redis 客户端的更多信息：<https://redis.io/topics/clients>。

- 有关 TCP Keepalive 的详细信息，请参阅相关页面：http://www.tldp.org/HOWTO/html_single/-TCP-Keepalive-HOWTO/。
- 有关 Linux 套接字的详细信息，请参阅有关的帮助页面：<http://man7.org/linux/man-pages/man7/-socket.7.html>和<http://man7.org/linux/man-pages/man2/setsockopt.2.html>。

8.5 配置内存策略

作为一个基于内存的数据存储，Redis 使用的内存空间比总是把数据持久化到磁盘中的数据库要多。虽然当下内存已经比较廉价了，但在生产环境中仍然需要对 Redis 占用的内存空间精打细算。此外，由于 Redis 经常被用作缓存，所以除了设置超时时间使键自动失效外，我们还需要考虑在缓存满时的淘汰策略。在本案例中，我们将介绍 Redis 中两个重要的内存配置参数。

8.5.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

8.5.2 操作步骤

接下来，让我们按照以下的步骤来学习如何配置内存策略：

1. 使用 INFO MEMORY 命令获取当前内存的使用情况：

```
127.0.0.1:6379> INFO MEMORY
# Memory
used_memory:836848
used_memory_human:817.23K
used_memory_rss:10174464
used_memory_rss_human:9.70M
used_memory_peak:973056
used_memory_peak_human:950.25K
used_memory_peak_perc:86.00%
used_memory_overhead:816270
used_memory_startup:765632
used_memory_dataset:20578
used_memory_dataset_perc:28.90%
total_system_memory:8371417088
total_system_memory_human:7.80G
```

```

used_memory_lua:37888
used_memory_lua_human:37.00K
maxmemory:0
maxmemory_human:0B
maxmemory_policy:noeviction
mem_fragmentation_ratio:12.16
mem_allocator:jemalloc-4.0.3
active_defrag_running:0
lazyfree_pending_objects:0

```

2. 出于演示的目的, 让我们把 MAXMEMORY 设置得比 used_memory 稍微大一点点:

```

127.0.0.1:6379> CONFIG SET MAXMEMORY 836900
OK

```

3. 尝试添加一些新键:

```

127.0.0.1:6379> SET new_key 1234567890
(error) OOM command not allowed when used memory > 'maxmemory'.

```

4. 将 MAXMEMORY-POLICY 改为 allkeys-lru:

```

127.0.0.1:6379> CONFIG SET MAXMEMORY-POLICY allkeys-lru
OK

```

5. 再次尝试添加一个新的键:

```

127.0.0.1:6379> SET new_key 1234567890
OK

```

8.5.3 工作原理

INFO MEMORY 命令输出了当前 Redis 实例中所有与内存相关的信息。我们感兴趣的内容是 used_memory、maxmemory 和 maxmemory_policy。used_memory 表示当前以字节为单位分配的内存空间。

在本例中, 共分配了 836848 字节的内存。maxmemory 是 Redis 的内存空间限制, 默认值为 0, 表示没有限制, 即 Redis 可以使用主机上所有可能的内存空间 (64 位系统没有限制, 而 32 位系统的限制是 3GB)。这个值可以通过 maxmemory 参数进行配置。maxmemory_policy 是在到达内存空间限制时的淘汰策略, 默认值是 noeviction, 即不淘汰键。我们可以使用 maxmemory-policy 参数配置淘汰策略。

在第 2 步中，我们将 `maxmemory` 设置成了略微大于 `used_memory` (836848) 的值 (836900)。这意味着可以为 Redis 分配的最大内存空间是 836900 字节（不包括 AOF 缓冲区的大小和从实例输出缓冲区的大小）。当我们试图创建新的键时，这个限制会很容易达到。当达到 `maxmemory` 的限制时 (`used_memory - AOF_buffer_size - slave_output_buffer_size >= maxmemory`) 且收到一个需要更多内存空间的请求时，Redis 将检查 `maxmemory-policy` 来决定如何应对。在本例中，因为我们设置的策略是 `noeviction`，所以 Redis 拒绝了键的创建请求并返回了一个错误。

表 8.1 列出了 `maxmemory-policy` 参数支持的选项，以及在达到 `maxmemory` 限制时相应的动作。

表 8.1 `maxmemory-policy` 参数

选项	动作
<code>noeviction</code>	不淘汰，只在写入操作时返回错误（DEL 和其他不需要更多内存空间的命令除外）
<code>allkeys-lru</code>	使用最近最少使用（ Least Recently Used, LRU ）算法淘汰任意键
<code>volatile-lru</code>	用近似的 LRU 算法淘汰设置了超时时间的键。如果没有这样的键则使用 <code>noeviction</code> 策略
<code>allkeys-lfu</code>	使用最不常用（ Least Frequently Used, LFU ）算法淘汰任意键
<code>volatile-lfu</code>	使用近似的 LFU 算法淘汰设置了超时时间的键。如果没有这样的键则使用 <code>noeviction</code> 策略
<code>allkeys-random</code>	随机淘汰键，可能是任意的键
<code>volatile-random</code>	随机淘汰设置了超时时间的键。如果没有这样的键则使用 <code>noeviction</code> 策略
<code>volatile-ttl</code>	淘汰马上就要过期的键（TTL 最小）。如果没有这样的键则使用 <code>noeviction</code> 策略

在第 4 步中，我们将 `maxmemory-policy` 改为 `allkeys-lru`，以便在达到 `maxmemory` 的限制时允许键被淘汰。因此，在那之后就可以成功创建一个新的键了。

8.5.4 更多细节

我们建议在生产环境的 Redis 服务器上配置 `maxmemory`，当在同一台主机上部署多个 Redis 实例时这非常有用且重要。每个实例在内存使用上是独立的，而且不会受到其他实例的影响。

值得一提的是，在 `used_memory` 值中也计入了客户端缓冲区占用的内存空间。因此，我们不应该期望数据对象能够使用全部的内存空间。值得一提的是，在计算 `maxmemory` 的上限时，从实例输出客户端缓冲区和 AOF 缓冲区的大小并没有被计入。

另外，我们不建议将 `maxmemory` 值设得太接近系统可用的内存大小，因为内存空间还应该留给其他进程，比如 Redis 自己就需要在进行 RDB 转储时创建子进程。

如果想把 Redis 作为缓存服务器使用，那么要务必将 `maxmemory-policy` 设为除 `noeviction` 外的值。不过，我们应该避免让 Redis 频繁地淘汰键，因为键的淘汰过程会严重影响服务器的性能。

8.5.5 相关内容

- Redis 中的 LRU 和 LFU 算法都是近似的，因为精确的实现需要消耗更多的内存。对于上述两个算法，有几个选项可以进行调优。读者可以参阅如下链接学习更多内容：<https://redis.io/topics/lru-cache>。

8.6 基准测试

如前所述，Redis 的处理速度取决于以下几个因素：CPU 性能、网络带宽、数据集大小、执行的操作等。因此，了解在部署到生产环境后 Redis 实例的速度到底有多快是非常重要的，而这就涉及到基准性能测试了。如果基准测试的结果不能满足我们的要求，那么就需要考虑升级硬件或调整使用 Redis 的方式。在本案例中，我们将介绍 Redis 的基准测试工具 `redis-benchmark`，并解释如何使用它进行基准测试。

8.6.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

8.6.2 操作步骤

接下来，让我们按照以下的步骤来学习如何进行基准测试。

1. 我们可以在 `bin/` 目录中找到 Redis 基准测试工具的可执行文件：

```
$ ls bin/redis-benchmark
bin/redis-benchmark
```

2. 要获得该工具的帮助信息，只需使用 `--help` 选项：

```
$ bin/redis-benchmark --help
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests>]
      [-k <boolean>]
```

```
-h <hostname>      Server hostname (default 127.0.0.1)
-p <port>           Server port (default 6379)
-s <socket>         Server socket (overrides host and port)
...
names are the same as the ones produced as output.
-I                 Idle mode. Just open N idle connections and wait.
```

3. 使用默认配置开始基准测试：

```
$ bin/redis-benchmark
===== PING_INLINE =====
100000 requests completed in 0.52 seconds
50 parallel clients
3 bytes payload
keep alive: 1

100.00% <= 0 milliseconds
190839.70 requests per second
...
===== MSET (10 keys) =====
100000 requests completed in 0.71 seconds
50 parallel clients
3 bytes payload
keep alive: 1

99.95% <= 1 milliseconds
100.00% <= 1 milliseconds
141043.72 requests per second
```

4. 我们还可以只针对一个 API 运行基准测试，并指定基准测试使用的数据量大小：

```
$ bin/redis-benchmark -t SET -c 100 -n 10000000 -r 10000000 -d 256
===== SET =====
10000000 requests completed in 73.63 seconds
100 parallel clients
256 bytes payload
keep alive: 1
```

```

99.18% <= 1 milliseconds
...
100.00% <= 4901 milliseconds
100.00% <= 4902 milliseconds
100.00% <= 4909 milliseconds
100.00% <= 4910 milliseconds
100.00% <= 4910 milliseconds
135806.83 requests per second

```

```

redis@gnu-hpc-desktop:~/bin$ ./redis-benchmark -n 100000 -q script load "redis.
    call('set','foo','bar')"
script load redis.call('set','foo','bar'): 190476.20 requests per second

```

5. 要测试利用管道功能可以获得多少性能提升，可以使用 -P 选项进行基准测试：

```

$ bin/redis-benchmark -t SET -c 100 -n 10000000 -r 10000000 -d 256 -P 10000
===== SET =====

```

```

10000000 requests completed in 36.21 seconds
100 parallel clients
256 bytes payload
keep alive: 1

```

```

0.00% <= 9 milliseconds
0.70% <= 10 milliseconds
2.50% <= 11 milliseconds
...
99.80% <= 478 milliseconds
99.90% <= 725 milliseconds
100.00% <= 725 milliseconds
276189.69 requests per second

```

6. 要进行 Lua 脚本的基准测试，可以使用 script load 选项：

```

$ bin/redis-benchmark -n 100000 script load "redis.call('set','foo','bar')"
===== script load redis.call('set','foo','bar') =====

```

```

100000 requests completed in 0.56 seconds
50 parallel clients
3 bytes payload
keep alive: 1

```



```
100.00% <= 0 milliseconds
179211.45 requests per second
```

8.6.3 工作原理

这个基准测试工具的使用是非常直观的。我们从基准测试结果中能获得的最重要的信息是：对于特定的 Redis 操作，Redis 实例每秒能够处理多少个请求。例如，从上一节的第一个基准测试中，我们可以看出 `MSET` 操作的处理速度是每秒 141043.72 个请求。

8.6.4 更多细节

请务必注意，基准测试结果仅仅给我们提供了一个极端的测试结果，我们永远不应该将其视作整个应用性能的压力测试（*stress test*）。

此外，如果想模拟真实的应用环境，我们应该在应用部署的那台主机上使用基准测试工具。这种方式可以使网络跳数对于基准工具和应用来说保持开销一致，对我们获得更真实的测试结果会有很大帮助。最后，如果我们想要对一个具有 N 个分片的 Redis 集群进行基准测试，那么集群的性能可以通过 N 个分片之一的基准测试结果乘以 N 进行粗略地估计（参见本章相关内容中 Redis 项目 GitHub 中的 Issue）。

8.6.5 相关内容

- 关于 Redis 基准工具的更多信息，请参阅：<https://redis.io/topics/benchmarks>。
- 在 Redis 项目的 Issue 板块中有一个关于 Redis 集群基准测试的讨论，请参阅：<https://github.com/antirez/redis/issues/4041>。
- 还有一个来自 RedisLabs 的基准测试工具 `memtier_benchmark`，如果读者乐意的话可以试用：https://github.com/RedisLabs/memtier_benchmark。

8.7 日志

在生产系统中，日志反映了过去的状态，所以是非常重要的。当系统发生故障时，我们可以通过查看和分析日志来找到问题发生的根本原因。在本案例中，我们将介绍 Redis 中日志相关的选项，并学习如何解读 Redis 服务器和 Sentinel 的日志。

8.7.1 准备工作

我们需要按照第 1 章开始使用 *Redis* 中下载和安装 *Redis* 的案例所描述的步骤安装一个 *Redis* 服务器。如果我们想查看 *Redis* 从实例和 *Sentinel* 的日志，那么还需要按照第 7 章配置高可用和集群中配置 *Sentinel* 的案例所描述的步骤配置一个带有 *Sentinel* 的环境。

8.7.2 操作步骤

接下来，让我们按照以下的步骤来学习如何配置日志选项及解读日志。

1. 我们可以通过修改配置文件中的 `loglevel` 选项来修改 *Redis* 的日志级别。在这里，我们将日志级别修改为 `debug`：

```
loglevel debug
```

2. 我们可以通过修改配置文件中的 `logfile` 选项修改日志文件的位置：

```
logfile "/var/log/redis/redis-server.log"
```

3. 我们还可以使用 `CONFIG SET` 命令来实时地修改 `loglevel` 和 `logfile`。这在我们希望临时降低日志级别以调试问题时很有用：

```
127.0.0.1:6379> CONFIG SET loglevel debug
```

```
OK
```

```
127.0.0.1:6379> CONFIG SET logfile "/var/log/redis/redis-server.log"
```

```
OK
```

4. 对其中一个 *Sentinel* 进行相同的操作，但指定一个不同的日志路径：

```
loglevel debug
```

```
logfile "/var/log/redis/redis-sentinel.log"
```

5. 查看 *Redis* 主实例的 `log` 文件：

```
$ less /var/log/redis/redis-server.log
```

```
1580:M 19 Nov 15:39:39.120 * Ready to accept connections
```

```
1580:M 19 Nov 15:39:39.120 - DB 0: 8 keys (0 volatile) in 8 slots HT.
```

```
1580:M 19 Nov 15:39:39.120 - 0 clients connected (0 slaves), 766144 bytes in
```

```
use
```

```
...
```

```
1580:M 19 Nov 15:39:39.502 * Starting BGSAVE for SYNC with target: disk
```

```

1580:M 19 Nov 15:39:39.503 * Background saving started by pid 1584
1584:C 19 Nov 15:39:39.507 * DB saved on disk
...
1580:M 19 Nov 15:40:06.829 # User requested shutdown...
1580:M 19 Nov 15:40:06.829 * Saving the final RDB snapshot before exiting.
1580:M 19 Nov 15:40:06.831 * DB saved on disk
...

```

6. 查看 Redis 从实例的 log 文件:

```

1645:S 19 Nov 15:46:22.916 * Connecting to MASTER 192.168.0.33:6379
1645:S 19 Nov 15:46:22.917 * MASTER <-> SLAVE sync started
1645:S 19 Nov 15:46:22.917 # Error condition on socket for SYNC: Connection
      refused
1645:S 19 Nov 15:46:23.926 * Connecting to MASTER 192.168.0.33:6379
1645:S 19 Nov 15:46:23.926 * MASTER <-> SLAVE sync started
1645:S 19 Nov 15:46:23.926 * Non blocking connect for SYNC fired the event.
1645:S 19 Nov 15:46:23.926 * Master replied to PING, replication can continue
...

```

7. 查看 Redis Sentinel 的 log 文件:

```

1710:X 19 Nov 15:48:18.014 # Sentinel ID is 3
      ef95f7fd6420bfe22e38bfded1399382a63ce5b
1710:X 19 Nov 15:48:18.014 # +monitor master mymaster 192.168.0.33 6379 quorum
      2
1710:X 19 Nov 15:48:18.513 - Accepted 192.168.0.33:37898
...

```

8.7.3 工作原理

Redis 从低到高共有四个日志级别: debug、verbose、notice 和 warning。只有等于或高于配置级别的消息才会被追加到日志中。例如,如果 loglevel 设为 verbose,则只会在日志中出现 verbose、notice 和 warning 级别的消息。我们设置的日志级别越低,在日志中看到的消息就越多。

在本案例中,我们将主实例和其中一个哨兵的日志级别设为了 debug。由于 debug 级别会输出所有的日志消息,这对于生产环境来说不是必需的。除非我们需要更多的日志来进行调试,否则默认的 notice 级别在大多数情况下是足够的。logfile 选项用于控制日志文件的位置。如果将其设置为空字符串,那么 Redis 会将日志输出到标准输出。

如果 Redis 启用了守护进程模式且 logfile 被设置为空字符串，那么所有的日志将被重定向到/dev/null 并被丢弃。

因此，当 Redis 在守护进程模式下运行时，必须正确地设置 logfile。通过查看 logfile，我们会发现 Redis 日志中的每一行都遵循如下的格式：

```
pid:role timestamp loglevel message
```

pid 是 Redis 服务器或 Sentinel 的进程 PID。role 是 Redis 实例的角色，由下列的单个字符表示：

```
X Sentinel
C RDB/AOF writing child
S slave
M master
```

timestamp 是事件的时间戳。loglevel 是由下列单个字符表示的日志级别：

```
. debug
- verbose
* notice
# warning
```

第 9 章

管理 Redis

在本章中，我们将学习下列案例：

- 管理 Redis 服务器配置。
- 使用 bin/redis-cli 操作 Redis。
- 备份和恢复。
- 监控内存使用情况。
- 管理客户端。
- 数据迁移。

9.1 本章概要

在 Redis 服务上线后，我们就需要进行日常的 Redis 运维操作了。在本章中，我们将重点介绍 Redis 服务器的管理。

我们将首先学习 Redis 服务器的配置。然后，我们将介绍使用最广泛的管理工具 `redis-cli` 的一些有用的功能。之后，我们将详细讨论数据的备份和恢复。接着，我们将学习如何通过多种指标来监控 Redis 实例的内存使用情况。作为 Redis 管理的另一个重要组成部分，我们还将讨论如何管理客户端。最后，我们将介绍一些用于在两个 Redis 单实例之间或 Redis 单实例与 Redis 集群之间进行数据迁移的工具。

9.2 管理 Redis 服务器配置

我们已经在之前的案例中多次设置或修改了 Redis 服务器配置。在本案例中，我们将再次讨论管理 Redis 服务器配置的方法，并介绍与 Redis 配置相关的两个命令。

9.2.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

9.2.2 操作步骤

接下来，让我们按照以下的步骤来学习如何管理 Redis 服务器配置。

1. 要从 config 文件中加载配置并启动一个 Redis 服务器，只需要在 `redis-server` 后追加配置文件的路径即可：

```
$ bin/redis-server conf/redis.conf
```

2. 我们可以在 Redis 配置文件中 `include` 指令来包含另一个配置文件。例如，我们可以将共享的配置放入 `/redis/conf/redis-common.conf` 并在 `redis.conf` 中引用：

```
include /redis/conf/redis-common.conf
```

3. 我们可以在 `redis-server` 后追加配置项的名称和对应的值，以在启动 Redis 服务器时覆盖配置参数：

```
$ bin/redis-server conf/redis.conf --loglevel verbose --port 6666
```

4. 使用 `CONFIG GET` 命令获取服务器的配置参数：

```
127.0.0.1:6379> CONFIG GET port
```

```
1) "port"
```

```
2) "6379"
```

5. 使用 `CONFIG SET` 命令设置或修改服务器的配置参数：

```
127.0.0.1:6379> CONFIG SET loglevel debug
```

```
OK
```

6. 使用 `CONFIG REWRITE` 命令把当前的服务器配置持久化到配置文件中：


```
127.0.0.1:6379> CONFIG SET loglevel debug
OK
127.0.0.1:6379> CONFIG REWRITE
OK
$ grep loglevel conf/redis.conf
loglevel debug
```

7. 使用 CONFIG RESETSTAT 命令重置 INFO 命令报告的统计项：

```
127.0.0.1:6379> INFO STATS
# Stats
total_connections_received:1
total_commands_processed:11
...
127.0.0.1:6379> CONFIG RESETSTAT
OK
127.0.0.1:6379> INFO STATS
# Stats
total_connections_received:0
total_commands_processed:1
...
```

9.2.3 工作原理

在第 1 步中，我们使用配置文件 `conf/redis.conf` 启动了一个 Redis 服务器。如果我们不指定配置文件，Redis 就会使用内置的默认配置启动。但是，完全采用默认配置应该只用于测试目的，而永远不应在生产环境中使用。

在第 2 步中，我们在 `redis.conf` 中包含了另一个配置文件 `redis-common.conf`。当多个 Redis 实例在同一个主机上运行时，这个功能是很有用的。我们可以把所有实例共享的配置放在 `redis-common.conf` 中，然后在每个实例的配置文件中将其包含。

`CONFIG REWRITE` 会将当前的配置写入配置文件中。如果我们使用 `CONFIG SET` 命令修改过配置，那么新的配置将被保存到配置文件中。在第 6 步中，我们首先修改了 `loglevel` 的配置，然后执行了 `CONFIG REWRITE` 命令。在那之后，我们可以看到 `redis.conf` 中的 `loglevel` 也被更新了。

请注意，如果 Redis 在启动时没有指定配置文件，那么 `CONFIG REWRITE` 命令会报错，因为它不知道向哪里写入配置。

CONFIG RESETSTAT 就像车辆里程表的重置按钮。该命令重置了由 INFO STATS 命令返回的几个计数器。具体来说，以下的计数器会被重置为 0：

- 键空间命中数（Keyspace hits）
- 键空间未命中数（Keyspace misses）
- 处理命令的数量（Number of commands processed）
- 收到连接的数量（Number of connections received）
- 过期键的数量（Number of expired keys）
- 拒绝连接的数量（Number of rejected connections）
- 上一次调用 fork(2) 的时间（Latest fork(2) time）
- aof_delayed_fsync 计数器（The aof_delayed_fsync counter）

9.3 使用 bin/redis-cli 操作 Redis

命令行工具 bin/redis-cli 对于操作 Redis 实例极为有用。在此前的案例中，我们已经演示了 redis-cli 的一些用法。在本案例中，我们将介绍 bin/redis-cli 的一些其他的实用操作。

9.3.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

9.3.2 操作步骤

接下来，让我们按照以下的步骤学习如何使用 redis-cli 操作 Redis。

1. 要使用 bin/redis-cli 运行一个命令，只需在 bin/redis-cli 后追加希望运行的命令即可：

```
$ bin/redis-cli HMSET hashkeyA field1 value1 feild2 value2
OK
```

2. 我们可以使用选项 --raw 以原始格式获取一个命令的输出：

```
$ bin/redis-cli --raw HGETALL hashkeyA
field1
value1
feild2
value2
```

注意

不使用选项`--raw`的话，我们在默认情况下会得到更具有可读性的输出：

```
$ bin/redis-cli hgetall hashkeyA
```

```
1) "field1"
```

```
2) "value1"
```

```
3) "feild2"
```

```
4) "value2"
```

3. 使用选项`--csv`以 CSV 格式获取一个命令的输出：

```
$ bin/redis-cli LPUSH listkeyA value1 value2 value3
```

```
(integer) 3
```

```
$ bin/redis-cli --csv LRange listkeyA 0 -1
```

```
"value3","value2","value1"
```

4. `bin/redis-cli` 命令的最后一个参数也可以从 `stdin` 中获取：

```
$ echo -n "bar" | bin/redis-cli -x SET foo
```

```
OK
```

```
$ bin/redis-cli GET foo
```

```
"bar"
```

```
$ bin/redis-cli -x SET hosts < /etc/hosts
```

```
OK
```

```
$ bin/redis-cli --raw GET hosts
```

```
127.0.0.1 localhost
```

```
127.0.1.1 gnuhpc-desktop
```

```
# The following lines are desirable for IPv6 capable hosts
```

```
::1 ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
```

```
ff00::0 ip6-mcastprefix
```

```
ff02::1 ip6-allnodes
```

```
ff02::2 ip6-allrouters
```

5. 如果我们想要执行一组保存在文件中的命令，那么可以使用管道符将文件内容重定向到

```
bin/redis-cli:
```

```
$ cat commands
```

```
set key1 value1
```

```
set key2 value2
```



```
$ cat commands |bin/redis-cli
```

```
OK
```

```
OK
```

6. 在使用管道案例中，我们已经了解到，如果要在 Redis 中执行大量的命令，那么应该利用管道功能来提高性能。我们可以使用 `--pipe` 选项在 `bin/redis-cli` 中启用管道功能：

```
$ unix2dos commands
```

```
unix2dos: converting file commands to DOS format ...
```

```
$ cat commands |bin/redis-cli --pipe
```

```
All data transferred. Waiting for the last reply...
```

```
Last reply received from server.
```

```
errors: 0, replies: 2
```

7. 使用 `-r <count>` 和 `-i <delay>` 选项以指定时间间隔重复地执行命令：

```
$ bin/redis-cli -r 5 -i 1 INFO MEMORY | grep used_memory:
```

```
used_memory:209036072
```

```
used_memory:209036072
```

```
used_memory:209036072
```

```
used_memory:209036072
```

```
used_memory:209036072
```

8. 使用 `--pattern` 选项遍历满足指定模式的键：

```
$ bin/redis-cli --scan --pattern 'session:*7ab0'
```

```
session:e9788147-a4dd-470d-af6b-df1e2fae7ab0
```

```
session:ef04be00-b4f2-4778-8023-82994d237ab0
```

```
session:4764e56f-a73e-4e64-bc68-03e51e067ab0
```

```
session:b030fb27-b560-4c39-8892-c5f6989f7ab0
```

```
session:51588697-191f-4e71-9a69-6540ff397ab0
```

```
session:37f0259c-f7ea-4fde-9883-ec3298e77ab0
```

```
session:c45db486-bfb2-4240-91b3-fd9850ba7ab0
```

```
session:7226bb09-b03a-4393-b98d-5f3f59097ab0
```

9.3.3 工作原理

前面的例子非常容易理解。我们可以很容易地将 `bin/redis-cli` 的输出用作另一个脚本的输入来进一步处理，反之亦然。

需要注意的一点是，使用 `--pipe` 选项时，我们必须将命令文件转换为 DOS 格式。这是因为，在启用管道功能时，`bin/redis-cli` 只能接受原始 Redis 协议格式的命令。对于 Redis 协议的细节，读者可以参考第 1 章开始使用 Redis 中理解 Redis 通信协议的案例。

9.3.4 更多细节

在 `bin/redis-cli` 中有几个性能相关的选项，我们将在第 10 章 Redis 故障诊断中介绍这些选项。

9.3.5 相关内容

- 有关 `bin/redis-cli` 的完整参考手册，请参阅官方文档：<https://redis.io/topics/rediscli>。

9.4 备份和恢复

按计划对生产数据库进行备份永远是很重要的。一旦诸如硬盘故障、意外数据删除或网络入侵之类的意外发生，备份就可以用来防止数据丢失。在第 6 章持久化中，我们已经了解到 Redis 可以将数据持久化到 RDB 文件中。在本案例中，我们将讨论 RDB 持久化选项并学习备份和恢复 Redis 数据的详细步骤。

9.4.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

9.4.2 操作步骤

接下来，让我们学习如何备份和恢复数据。

备份 Redis 数据

首先，按以下步骤对 Redis 数据进行备份：

1. 在 Redis 服务器上执行 `BGSAVE` 命令：

```
$ bin/redis-cli BGSAVE
Background saving started
```

2. 将生成的 RDB 文件复制到安全的位置：

```
$ cp /var/lib/redis/dump.rdb /mnt/backup/redis/dump.$(date +%Y%m%d%H%M).rdb
```

从 RDB 文件恢复 Redis 数据

接下来，按以下步骤从上一节生成的 RDB 文件中恢复 Redis 数据。

1. 检查 Redis 是否启用了 AOF：

```
$ bin/redis-cli CONFIG GET appendonly
```

```
1) "appendonly"
```

```
2) "yes"
```

2. 如果没有启用 AOF，则跳过这一步，直接进行第 3 步。否则，先禁用 AOF：

```
$ bin/redis-cli CONFIG SET appendonly no
```

```
OK
```

```
$ bin/redis-cli CONFIG REWRITE
```

```
OK
```

3. 关闭 Redis 服务器，并将数据目录中的 AOF 和 RDB 文件删除或重命名：

```
$ bin/redis-cli SHUTDOWN
```

```
$ rm *.aof *.rdb
```

4. 将要恢复的 RDB 快照文件复制到 Redis 的数据目录中，并将其重命名为 dump.rdb（确保与配置中的 dbfilename 匹配）：

```
$ cp /mnt/backup/redis/dump.201712250100.rdb /var/lib/redis/dump.rdb
```

5. 给 dump.rdb 设置正确权限，并启动 Redis 服务器：

```
$ chown redis:redis dump.rdb
```

```
$ bin/redis-server conf/redis-server.conf
```

6. 如有必要，重新启用 AOF 持久化：

```
$ bin/redis-cli CONFIG SET appendonly yes
```

```
OK
```

```
$ bin/redis-cli CONFIG REWRITE
```

```
OK
```


9.4.3 工作原理

上述步骤非常简单易行。BGSAVE 命令将所有数据保存到了 RDB 文件中，因此只要备份这个文件就可以了。通常，我们需要配置一个 crontab 作业来定期地进行备份。

注意，在启用 AOF 时，Redis 会首先试图从 AOF 文件中恢复数据，所以在将数据恢复到 Redis 之前禁用 AOF 是很重要的。如果此时找不到 AOF 文件（我们在第 3 步中将其进行了删除/重命名），Redis 将会从一个空数据集开始恢复。在这种情况下，一旦键的更改触发了 RDB 快照，原始的 RDB 文件也将被重写。

9.5 监控内存使用情况

由于 Redis 是一个基于内存的数据服务，所以监控 Redis 的内存使用是非常重要的。在本案例中，我们将学习如何通过 INFO 命令和 MEMORY 命令来监控 Redis 的内存使用情况。

9.5.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

9.5.2 操作步骤

接下来，让我们按照以下的步骤来学习如何监控 Redis 的内存使用情况。

1. 我们可以通过 `bin/redis-cli` 执行 `INFO MEMORY` 命令来获取 Redis 内存相关的总体指标：

```
$ bin/redis-cli INFO MEMORY
# Memory
used_memory:211428088
used_memory_human:201.63M
used_memory_rss:251547648
used_memory_rss_human:239.89M
used_memory_peak:3865330064
used_memory_peak_human:3.60G
used_memory_peak_perc:5.47%
used_memory_overhead:49242362
used_memory_startup:765624
used_memory_dataset:162185726
```

```

used_memory_dataset_perc:76.99%
total_system_memory:67467202560
total_system_memory_human:62.83G
used_memory_lua:40960
used_memory_lua_human:40.00K
maxmemory:4000000000
maxmemory_human:3.73G
maxmemory_policy:noeviction
mem_fragmentation_ratio:1.19
mem_allocator:jemalloc-4.0.3
active_defrag_running:0
lazyfree_pending_objects:0

```

2. 我们可以使用 `MEMORY USAGE` 命令来估算一个键的内存使用情况：

```

127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> MEMORY USAGE foo
(integer) 54

```

注意

`MEMORY USAGE` 命令没有计入键本身的长度和键过期信息所消耗的内存：

```

127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> MEMORY USAGE foo
(integer) 54
127.0.0.1:6379> set foooooooooooooooooooooo bar
OK
127.0.0.1:6379> MEMORY USAGE foooooooooooooooooooooo
(integer) 54
4) "value2"
127.0.0.1:6379> EXPIRE foo 600
(integer) 1
127.0.0.1:6379> MEMORY USAGE foo
(integer) 54

```

3. 我们可以使用 `MEMORY STATS` 命令来查看 Redis 实例中各部分的内存消耗情况：

```
$ bin/redis-cli --csv MEMORY STATS
"peak.allocated",3865330064,"total.allocated",211428088,"startup.allocated",
765624,"replication.backlog",0,"clients.slaves",0,"clients.normal",
83346,"aof.buffer",0,"db.0","overhead.hashtable.main",48393288,"overhead.
.hashtable.expires",32,"db.1","overhead.hashtable.main",72,"overhead.
hashtable.expires",0,"overhead.total",49242362,"keys.count",1000118,"keys.
bytes-per-key",210,"dataset.bytes",162185726,"dataset.percentage",
"76.988433837890625","peak.percentage",5.4698591232299805,"
fragmentation",1.1897543668746948"
```

4. Redis 还为定位内存问题提供了一个强大的诊断命令 MEMORY DOCTOR:

```
$ bin/redis-cli MEMORY DOCTOR
```

Sam, I detected a few issues in this Redis instance memory implants:

* Peak memory: In the past this instance used more than 150% the memory that is currently using. The allocator is normally not able to release memory after a peak, so you can expect to see a big fragmentation ratio, however this is actually harmless and is only due to the memory peak, and if the Redis instance Resident Set Size (RSS) is currently bigger than expected, the memory will be used as soon as you fill the Redis instance with more data. If the memory peak was only occasional and you want to try to reclaim memory, please try the MEMORY PURGE command, otherwise the only other option is to shutdown and restart the instance.

I'm here to keep you safe, Sam. I want to help you.

5. 我们还可以使用 bin/redis-cli 的 --bigkeys 选项来查找最大的键和每种类型的键的平均大小:

```
$ bin/redis-cli --bigkeys
# Scanning the entire keyspace to find biggest keys as well as
# average sizes per key type. You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).
[00.00%] Biggest hash found so far 'session:39b2276f-1474-4afe-a619-2929
b91d0430' with 7 fields
[00.00%] Biggest hash found so far 'session:bd9ff9f7-0e4f-42e3-9e08-0179120
b0e22' with 10 fields
```



```
[00.30%] Biggest string found so far 'lru:31' with 5 bytes
[01.98%] Biggest string found so far 'key2' with 6 bytes
[26.47%] Biggest string found so far 'name' with 8 bytes
[27.31%] Biggest hash found so far 'user:0000004' with 1000 fields
[39.66%] Biggest string found so far 'hosts' with 351 bytes
[56.95%] Biggest hash found so far 'bighash' with 18593 fields
[87.13%] Biggest list found so far 'listkeyA' with 3 items
[99.99%] Sampled 1000000 keys so far
----- summary -----
Sampled 1000116 keys in the keyspace!
Total key length in bytes is 44000724 (avg len 44.00)
Biggest string found 'hosts' has 351 bytes
Biggest list found 'listkeyA' has 3 items
Biggest hash found 'bighash' has 18593 fields
103 strings with 862 bytes (00.01% of keys, avg size 8.37)
1 lists with 3 items (00.00% of keys, avg size 3.00)
0 sets with 0 members (00.00% of keys, avg size 0.00)
1000012 hashes with 5528165 fields (99.99% of keys, avg size 5.53)
0 zsets with 0 members (00.00% of keys, avg size 0.00)
```

9.5.3 工作原理

对于获取整个 Redis 实例内存使用情况来说，INFO MEMORY 命令是使用最广泛的。下面是该命令返回的各个指标的含义：

- `used_memory`: Redis 使用分配器分配的总字节数。
- `used_memory_human`: 前一个值的用户可读性好的形式。
- `used_memory_rss`: 从操作系统的角度看 Redis 分配的字节数。
- `used_memory_rss_human`: 前一个值的用户可读性好的形式。
- `used_memory_peak`: Redis 消耗内存的峰值（以字节为单位）。
- `used_memory_peak_human`: 前一个值的用户可读性好的形式。
- `used_memory_peak_perc`: $(\text{used_memory} / \text{used_memory_peak}) * 100\%$ 。
- `used_memory_overhead`: Redis 为了维护数据集的内部机制所需的内存开销，包括所有客户端输出缓冲区、查询缓冲区、AOF 重写缓冲区和主从复制的 backlog。
- `used_memory_startup`: 当 Redis 服务器启动时消耗的内存。
- `used_memory_dataset`: `used_memory - used_memory_overhead`。

- `used_memory_dataset_perc:100% * (used_memory_dataset / (used_memory - used_memory_startup))`。
- `total_system_memory`: 整个系统的内存。
- `total_system_memory_human`: 前一个值的用户可读性好的形式。
- `used_memory_lua`: Lua 脚本存储占用的内存。
- `used_memory_lua_human`: 前一个值的用户可读性好的形式。
- `maxmemory`: Redis 实例的最大内存配置。
- `maxmemory_human`: 前一个值的用户可读性好的形式。
- `maxmemory_policy`: 当达到 `maxmemory` 时的淘汰策略。
- `mem_fragmentation_ratio`: `used_memory_rss/used_memory`。
- `mem_allocator`: 内存分配器。
- `active_defrag_running`: 0 表示没有活动的 defrag 任务正在运行, 1 表示有活动的 defrag 任务正在运行 (译者注: defrag 指内存碎片整理)。
- `lazyfree_pending_objects`: 0 表示不存在延迟释放 (译者注: 也有资料译为惰性删除) 的挂起对象。

`MEMORY USAGE` 命令用于估算一个键消耗的内存。该命令有一个选项, 用于指定进行估算时采样的数量。在底层, Redis 利用来自一个数据类型 (`hash`、`list` 等) 集合中的平均样本来粗略地估算键的内存使用情况。如下所示, 我们取的样本越多, 结果越准确。不过, 更多的样本和更高的数据准确性是以更慢的估算过程为代价的:

```
$ cat populatedata6.sh
#!/bin/bash
DATAFILE="hash.data"
echo -n "HMSET bighash " > $DATAFILE
for i in `seq -f "%010g" 1 30000`
do
echo -n "$[ $RANDOM % 30000] $[ $RANDOM % 30000] " >> $DATAFILE
done
unix2dos $DATAFILE
cat $DATAFILE | bin/redis-cli
$ bash populatedata6.sh
unix2dos: converting file hash.data to DOS format ...
OK
$ time bin/redis-cli MEMORY USAGE bighash SAMPLES 100
(integer) 1084090
```

```

real 0m0.002s
user 0m0.000s
sys 0m0.000s
$ time bin/redis-cli MEMORY USAGE bighash SAMPLES 15000
(integer) 1083861
real 0m0.003s
user 0m0.000s
sys 0m0.004s
$ time bin/redis-cli MEMORY USAGE bighash SAMPLES 20000
(integer) 1083868
real 0m0.004s
user 0m0.000s
sys 0m0.000s

```

MEMORY STATS 命令显示了 Redis 中内存消耗的每个部分，其中有一些指标与 MEMORY INFO 命令的输出有交叉。我们接下来只学习 MEMORY STATS 命令返回的特有指标的含义：

- replication.backlog: 主从复制 backlog 的内存消耗（以字节为单位）。
- clients.slaves: 从实例客户端输出缓冲区的内存消耗（以字节为单位）。
- clients.normal: 普通客户端缓冲区的内存消耗（以字节为单位）。
- aof.buffer: AOF 缓冲区的内存消耗（以字节为单位）。
- overhead.hashtable.main: 维护 Redis 中数据的内存开销。
- overhead.hashtable.expires: 在 Redis 中存储键过期信息的内存开销。
- keys.count: 键总数（包括 Redis 实例的所有数据库）。
- keys.bytes-per-key: $(total.allocated - startup.allocated) / keys.count$ 。

MEMORY DOCTOR 命令为某些内存问题提供了一些提示。该命令的主要逻辑如图 9-1 所示。

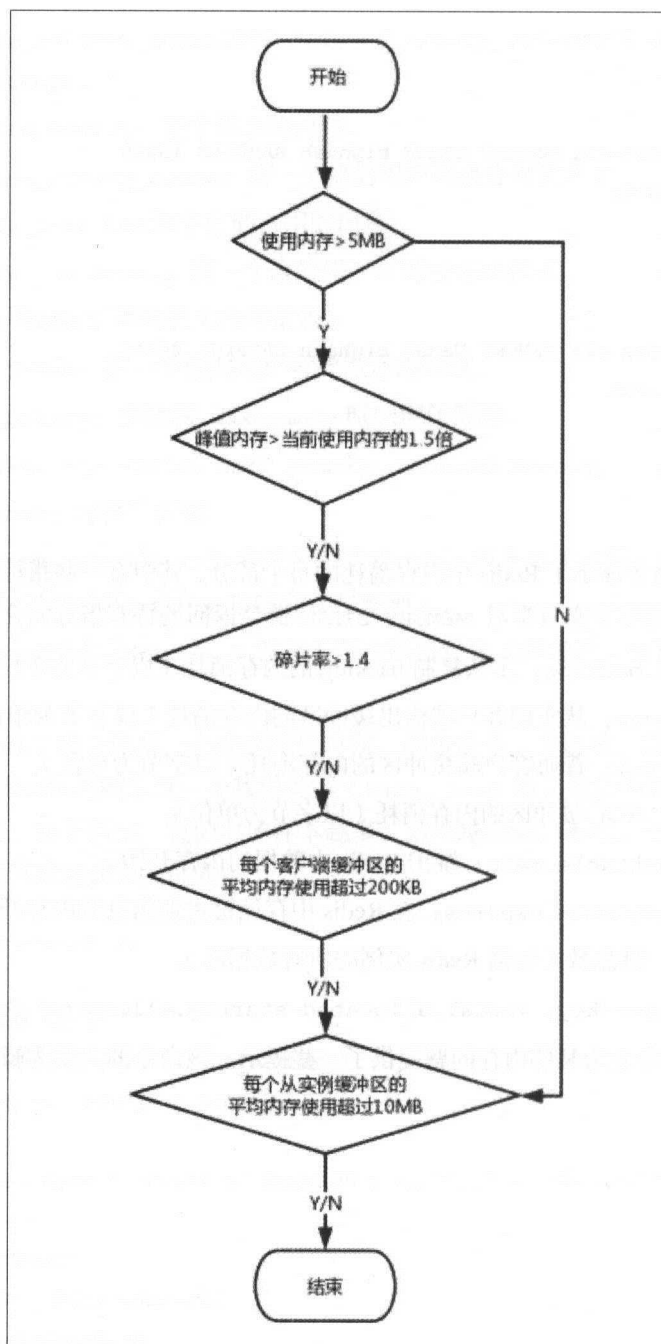


图 9.1 MEMORY DOCTOR 命令的主要逻辑

9.5.4 更多细节

在第 10 章 Redis 故障诊断中内存问题的故障诊断的案例中,我们将利用上一节介绍的命令来解决内存问题。

9.5.5 相关内容

- INFO MEMORY 命令的完整参考,请参阅: <https://redis.io/commands/INFO>。
- 4.0 版本后引入的 MEMORY 命令,请参阅 4.0 的发布说明: <http://antirez.com/news/110>。

9.6 管理客户端

从本质上说,Redis 是一个采用客户端—服务器模型的 TCP 服务器。因此,对于 Redis 的管理员来说,管理连接到 Redis 服务器的客户端是非常重要的。在本案例中,我们将首先介绍如何收集 Redis 实例的总体信息;然后,我们将学习如何在服务器端列出和杀死 Redis 客户端。我们还将学习一些重要的配置参数。

9.6.1 准备工作

我们需要按照第 5 章复制 (Replication) 中配置 Redis 复制机制的案例所描述的步骤完成 Redis 复制机制的配置。

9.6.2 操作步骤

接下来,让我们按照以下的步骤来学习如何管理客户端。

1. 出于演示目的,首先启动两个 redis-benchmark 进程:

```
bin$ nohup ./redis-benchmark -c 5 -n 100000 -r 1000 -d 1000 &  
bin$ nohup ./redis-benchmark -c 5 -n 100000 -r 1000 -d 1000 &
```

2. 然后,使用 bin/redis-cli 执行 BLPOP 命令,如下:

```
$ bin/redis-cli BRPOP job_queue 0
```

3. 使用 bin/redis-cli 执行 INFO CLIENTS 和 INFO STATS 命令来收集 Redis 客户端相关的指标:

```
$ bin/redis-cli INFO CLIENTS
# Clients
connected_clients:12
client_longest_output_list:28
client_biggest_input_buf:0
blocked_clients:1
$ bin/redis-cli INFO STATS |grep connection
total_connections_received:4921
rejected_connections:0
```

4. 如果读者想获取每个客户端的详细信息，可以使用 `bin/redis-cli` 执行 `CLIENT LIST` 命令：

```
127.0.0.1:6379> CLIENT LIST
id=4827 addr=127.0.0.1:47838 fd=7 name= age=0 idle=0 flags=N db=0 sub=0 psub=0
multi=-1 qbuf=0 qbuf-free=32768 obl=4 oll=0 omem=0 events=r cmd=hset
...
id=4826 addr=127.0.0.1:47836 fd=18 name= age=1 idle=0 flags=N db=0 sub=0 psub
=0 multi=-1 qbuf=0 qbuf-free=32768 obl=1009 oll=0 omem=0 events=r cmd=lpop
id=4 addr=127.0.0.1:34892 fd=8 name= age=352 idle=0 flags=S db=0 sub=0 psub=0
multi=-1 qbuf=0 qbuf-free=0 obl=3303 oll=0 omem=0 events=r cmd=repliconf
id=4036 addr=127.0.0.1:46254 fd=29 name= age=228 idle=209 flags=b db=0 sub=0
psub=0 multi=-1 qbuf=0 qbuf-free=0 obl=0 oll=0 omem=0 events=r cmd=brpop
id=5 addr=127.0.0.1:38192 fd=9 name= age=345 idle=0 flags=N db=0 sub=0 psub=0
multi=-1 qbuf=0 qbuf-free=32768 obl=0 oll=0 omem=0 events=r cmd=client
```

警告

使用 `CLIENT SETNAME` 命令通过设置客户端的名称来标识一个客户端。如果多个应用连接到同一个 Redis 实例，那么这样做能够更容易地区分连接：

```
127.0.0.1:6379> CLIENT SETNAME bin/redis-cli
OK
127.0.0.1:6379> CLIENT LIST
id=4902 addr=127.0.0.1:47990 fd=9 name=bin/redis-cli age=8 idle=0 flags=N
db=0 sub=0 psub=0 multi=-1 qbuf=0 qbuf-free=32768 obl=0 oll=0 omem=0 events=r
cmd=client
```

5. 通过 `CLIENT KILL client-host:client-port` 命令终止客户端的连接：


```
127.0.0.1:6379> CLIENT KILL 127.0.0.1:46254
```

```
OK
```

6. 在客户端，我们会发现连接被立即断开：

```
$ bin/redis-cli BRPOP job_queue 0
```

```
Error: Server closed the connection
```

警告

除了终止连接外，使用 `CLIENT PAUSE` 命令将所有的 Redis 客户端暂停指定的一段时间。

例如，我们可以将 Redis 连接暂停 60 秒：

```
127.0.0.1:6379> CLIENT PAUSE 60000
```

```
OK
```

```
127.0.0.1:6379> DBSIZE
```

```
(integer) 3001
```

```
(57.13s)。
```

在暂停期间，Redis 服务器会停止处理来自普通客户端和发布/订阅客户端的所有命令。所有已连接的客户端将在暂停期间被挂起，并且请求连接也将被挂起。不过，主从复制会继续进行。此外，在暂停期间，数据集保证不会发生改变。因此，我们可以在暂停期间使用复制机制进行数据迁移。

9.6.3 工作原理

`INFO CLIENTS` 命令返回了四个与客户端相关的指标。以下是每个指标的含义：

- `connected_clients`：客户端连接数。请注意，这个指标不包括来自从实例的连接。
- `client_longest_output_list`：当前客户端连接中最长的输出列表。我们将稍后在本案例中讨论这个指标。
- `client_biggest_input_buf`：当前客户端连接中最大的输入缓冲区。我们将稍后在本案例中讨论这个指标。
- `blocked_clients`：被挂起在一个诸如 `BLPOP`、`BRPOP`、`BRPOPLPUSH` 等命令上的客户端数量。即使这个指标不等于 0 也不用紧张；它只表示被阻塞命令挂起的客户端的数量。

`INFO STATS` 命令返回两个与客户端相关的指标。一个指标是 `total_connections_received`，即自 Redis 服务器启动以来的总连接数。另一个指标是 `rejected_connections`，表示由于 `maxclients` 的限制而被拒绝的连接数。我们应该特别注意这个指标；一旦这个指标的值增加，就意味着已经达到 `maxclients` 的限制，新的连接会被拒绝。

CLIENT LIST 命令显示了每个连接的详细信息。下面是一个连接所涉及所有字段的含义：

- id: 一个唯一的 64 位客户端 ID
- addr: 客户端的 IP 地址/端口
- fd: 对应套接字的文件描述符
- age: 连接的总持续时间（以秒为单位）
- idle: 连接的空闲时间（以秒为单位）
- flags: 客户端状态。例如，N 代表普通客户端，S 代表从实例，等等
- db: 当前的数据库 ID
- sub: 订阅的通道数
- psub: 以模式匹配方式订阅的通道数
- multi: 在 MULTI/EXEC 上下文中执行的命令数
- qbuf: 查询缓冲区长度（0 表示没有挂起的查询）
- qbuf-free: 查询缓冲区的空闲空间（0 表示缓冲区已满）
- obl: 输出固定缓冲区使用的空间
- oll: 输出列表长度（当缓冲区满时，响应在此列表中排队）
- omem: 输出缓冲区使用的空间
- events: 文件描述符事件（r 代表套接字可读，w 表示套接字可写）
- cmd: 客户端发出的最后一个命令

在前面的字段中，出现了几个新的概念。其中，第一个概念是**查询缓冲区（query buffer）**。对于每一个连接，Redis 会分配一块内存作为该连接的查询缓冲区。从该连接接收的所有命令将在执行之前都被保存在查询缓冲区中。Redis 服务器会从查询缓冲区中获取命令并执行命令。查询缓冲区的大小在 0 到 1GB 之间。一旦查询缓冲区的大小超过了 1GB，Redis 就会尽快将连接关闭。qbuf 和 qbuf-free 给出了这个缓冲区的大小和空闲空间。

另一个新概念是**输出缓冲区（output buffer）**。我们已经在复制机制的故障诊断案例中介绍了从实例输出缓冲区，所以这并不是一个全新的概念。实际上，每个客户端（包括从实例客户端）都有一个输出缓冲区。从 Redis 服务器返回的所有数据将首先被保存在这个缓冲区中，然后再发送给客户端。客户端输出缓冲分为两个部分：固定缓冲区（16KB）和动态输出列表。指标 obl 表示固定缓冲区使用的空间，指标 oll 表示动态输出列表中的内容数，指标 omem 表示该连接客户端输出缓冲区使用的空间（以字节为单位）。

最后，指标 age 和 idle 可以帮助我们发现长期空闲的连接。在大多数情况下，长时间空闲的连接很可能就是连接泄漏问题的信号。

9.6.4 更多细节

请注意，为查询缓冲区和输出缓冲区分配的内存是被计入由 `INFO MEMORY` 命令返回的 `used_memory_overhead` 中的，但不包括监控客户端的输出缓冲区。鉴于 `used_memory = used_memory_overhead + used_memory_dataset` 的事实，`used_memory` 指标也计入了这两种缓冲区。

9.6.5 相关内容

- 对于 Redis 客户端命令的完整参考手册，读者可以参阅官方文档并找到对 `CLIENT` 命令所有输出的说明：<https://redis.io/commands#server>
- 对于客户端的重要配置参数，读者可以参阅第 8 章生产环境部署中配置客户端连接选项的案例。

9.7 数据迁移

由于某些原因，我们可能需要移动或复制存储在一个 Redis 实例中的数据。这时，就需要进行数据迁移。在本案例中，我们将学习几种可以用来进行 Redis 数据迁移的方法。

9.7.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

我们还需要学习第 7 章配置高可用和集群中配置 Redis Cluster 的案例，来了解 Redis 集群的基本知识。

9.7.2 操作步骤

在单个 Redis 实例间进行数据迁移的第一种方法如下。

1. 首先，启动两个 Redis 实例：

```
$ bin/redis-server conf/redis.conf
$ bin/redis-server conf/redis2.conf
$ ps -ef |grep redis-server
redis 361 32707 0 14:14 pts/5 00:00:00 grep redis-server
redis 21413 1 0 Dec10 ? 00:03:10 bin/redis-server 0.0.0.0:6379
redis 32762 1 0 14:13 ? 00:00:00 bin/redis-server 0.0.0.0:6380
127.0.0.1:6379> info Keyspace
```



```
# Keyspace
db0:keys=1000000,expires=0,avg_ttl=922042600
```

2. 通过把目的实例设置为源实例的一个从实例，我们可以复制源 Redis 实例中的所有数据：

```
127.0.0.1:6380> SLAVEOF 192.168.1.7 6379
OK
127.0.0.1:6380> info Keyspace
# Keyspace
db0:keys=1000000,expires=0,avg_ttl=922042600
```

3. 当同步完成后，我们通过 SLAVEOF NO ONE 命令将目的实例设置为主实例：

```
127.0.0.1:6380> SLAVEOF NO ONE
OK
```

在单个 Redis 实例间进行数据迁移的第二种方法如下。

1. 除了设置主从关系外，我们还可以使用 AOF 持久化文件来进行数据迁移。源实例和目的实例都需要启用 AOF 持久化，并将 aof-use-rdb-preamble 参数设为 yes。这样做可以同时利用 RDB 和 AOF 的优点：

```
127.0.0.1:6379> CONFIG SET appendonly yes
OK
127.0.0.1:6379> CONFIG SET aof-use-rdb-preamble yes
OK
127.0.0.1:6380> CONFIG SET appendonly yes
OK
127.0.0.1:6380> CONFIG SET aof-use-rdb-preamble yes
OK
```

2. 在源实例中执行 BGREWRITEAOF 命令：

```
127.0.0.1:6379> BGREWRITEAOF
Background saving started
```

3. 停止目的实例：

```
$ bin/redis-cli -p 6380 shutdown
```

4. 将 append.aof 文件复制到目的实例的数据目录并重新启动目的实例。

在单个 Redis 实例间进行数据迁移的第三种方法是使用 MIGRATE 命令将特定的键从一个实例移动到另一个实例。

```
$ KEYS=`bin/redis-cli -p 6379 --scan --pattern user:* | awk '{printf("%s ",$1)}'`
$ echo $KEYS
user:0000004 user:0000008 user:0000009 user:0000007 user:0000003 user:0000005
user:0000002 user:0000006 user:0000000 user:0000010 user:0000001
$ bin/redis-cli -p 6379 MIGRATE 127.0.0.1 6380 "" 0 5000 REPLACE keys $KEYS
OK
$ bin/redis-cli -p 6380 --scan --pattern user:*
user:0000000
user:0000002
user:0000006
user:0000001
user:0000008
user:0000010
user:0000007
user:0000005
user:0000009
user:0000004
user:0000003
```

在单个 Redis 实例与一个 Redis 集群间进行数据迁移的第一种方法如下。

1. 要将数据从一个 Redis 实例迁移到一个全新的 Redis 集群，首先要做的是建立一个 Redis 集群，该集群中只有主实例且每个主实例都没有从实例。出于迁移的目的，我们必须将所有的槽移到一个主实例。要学习如何配置 Redis 集群，请参考第 7 章配置高可用和集群中配置 *Redis Cluster* 的案例。集群的最终状态形如：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 cluster nodes
e74f75b566fcfb21c3543cfb71ddc8c6df68b9d9 192.168.1.67:6381@16381 master -
0 1513216226912 6 connected
02f217d2fb77c068c462013851fa287de53d577c 192.168.1.66:6380@16380 master - 0
1513216226000 7 connected
610c014ce0478453ab478db6707839556bb05e13 192.168.1.65:6379@16379 myself,master
- 0 1513216224000 8 connected 0-16383
```

2. 启用 AOF 持久化，并将 `aof-use-rdb-preamble` 设为 `yes`：

```
127.0.0.1:6379> CONFIG SET appendonly yes
```

```
OK
```

```
127.0.0.1:6379> CONFIG SET aof-use-rdb-preamble yes
```

```
OK
```

3. 让我们生成一些测试数据，然后在后台进行 AOF 重写。我们可以在 `bin/redis-cli` 中使用 `INFO PERSISTENCE` 命令检查重写过程是否已经完成：

```
127.0.0.1:6379> DEBUG POPULATE 1000000
```

```
OK
```

```
127.0.0.1:6379> INFO KEYSpace
```

```
# Keyspace
```

```
db0:keys=1000000,expires=0,avg_ttl=0
```

```
127.0.0.1:6379> BGREWRITEAOF
```

```
Background saving started
```

```
127.0.0.1:6379> INFO PERSISTENCE
```

```
# Persistence
```

```
aof_rewrite_in_progress:0
```

4. 关闭拥有所有槽的主实例，然后将 AOF 持久化文件复制到这个 Redis 实例的数据目录：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 SHUTDOWN
```

```
redis@192.168.1.65:~/data> scp redis@192.168.1.7:/redis/appendonly.aof .
```

5. 启动这个 Redis 实例，并检查 AOF 文件是否已被加载：

```
redis@192.168.1.65:~> bin/redis-server conf/redis-6379.conf
```

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 INFO Keyspace
```

```
# Keyspace
```

```
db0:keys=1000000,expires=0,avg_ttl=0
```

6. 使用 `redis-trib.rb` 脚本对数据集重新分片：

```
redis@192.168.1.65:~> script/redis-trib.rb reshard --from 610
```

```
    c014ce0478453ab478db6707839556bb05e13 --to
```

```
    e74f75b566fcfb21c3543cfb71ddc8c6df68b9d9 --slots 5000 --yes
```

```
192.168.1.65:6379
```

```
redis@192.168.1.65:~> script/redis-trib.rb reshard --from 610
```

```
    c014ce0478453ab478db6707839556bb05e13 --to 02
```

```
    f217d2fb77c068c462013851fa287de53d577c --slots 5000 --yes
```

```
192.168.1.65:6379
```


7. 在集群中的一个节点上使用 `CLUSTER NODES` 命令检查是否所有的槽都已经按照我们的意愿进行了分片：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 cluster nodes
02f217d2fb77c068c462013851fa287de53d577c 192.168.1.66:6380@16380 master - 0
    1513158338408 7 connected 5000-9999
e74f75b566fcfb21c3543cfb71ddc8c6df68b9d9 192.168.1.67:6381@16381 master - 0
    1513158339410 6 connected 0-4999
610c014ce0478453ab478db6707839556bb05e13 192.168.1.65:6379@16379 myself,master
    - 0 1513158338000 5 connected 10000-16383
```

8. 我们还可以向集群的每个主节点发送 `DBSIZE` 命令。通过计算所有键 ($389,638 + 305,153 + 305,209 = 1,000,000$) 的数量总和，我们可以得出所有数据已被迁移到 Redis 集群的结论：

```
redis@192.168.1.65:~> script/redis-trib.rb call 192.168.1.65:6379 dbsize
>>> Calling DBSIZE
192.168.1.65:6379: 389638
192.168.1.67:6381: 305209
192.168.1.66:6380: 305153
```

9. 最后，作为必要的步骤，我们应该给每个主节点增加从节点作为备份。读者可以参考第 7 章配置高可用和集群中配置 *Redis Cluster* 的案例学习如何进行上述操作。

在单个 Redis 实例与一个 Redis 集群间进行数据迁移的第二种方法如下。

1. 首先，配置 Redis 集群中的所有主节点并完成槽的分配。该集群的状态如下所示：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 cluster nodes
6033a29badec100591bf42dee02a76d5abc66131 192.168.1.66:6380@16380 master - 0
    1513222558953 0 connected 5401-11000
1delle3729e66ff694820fa3e3d5de5e3e73f4d8 192.168.1.65:6379@16379 myself,master
    - 0 1513222557000 1 connected 0-5400
3e87c418b9a685bb47e08ce7eaec631a9f58c282 192.168.1.67:6381@16381 master - 0
    1513222559956 2 connected 11001-16383
```

2. 关闭集群中的一个节点，并将 AOF 持久化文件复制到节点的数据目录中。然后，重新启动节点：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 shutdown
redis@192.168.1.65:~> cd data
redis@192.168.1.65:~/data> scp redis@192.168.1.7:/redis/appendonly.aof .
```

```
redis@192.168.1.65:~> bin/redis-server conf/redis-6379.conf
```

3. 使用 `bin/redis-cli` 的 `-c` 选项连接到集群。当尝试获取位于在集群其他节点上的键时，将会报错：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 -c
192.168.1.65:6379> RANDOMKEY
"key:50375"
192.168.1.65:6379> get "key:50375"
-> Redirected to slot [6890] located at 192.168.1.66:6380
(nil)
```

4. 使用 `redis-trib.rb` 的 `fix` 命令修复这个问题。然后，在集群的每个节点上执行 `DBSIZE` 命令：

```
redis@192.168.1.65:~> script/redis-trib.rb fix 192.168.1.65:6379
redis@192.168.1.65:~> script/redis-trib.rb call 192.168.1.65:6379 dbsize
>>> Calling DBSIZE
192.168.1.65:6379: 329674
192.168.1.67:6381: 328536
192.168.1.66:6380: 341790
```

5. 现在，我们可以成功地获取键了：

```
redis@192.168.1.65:~> bin/redis-cli -h 192.168.1.65 -p 6379 -c
192.168.1.65:6379> get "key:50375"
-> Redirected to slot [6890] located at 192.168.1.66:6380
"value:50375"
```

提示

读者可以利用这种方法将数据从一个小的 Redis 集群迁移到另一个具有更多节点的集群。除了需要将多个 AOF 持久化文件复制到目标集群的不同节点外，步骤几乎是相同的。

除了复制持久化文件外，我们可以使用 `redis-trib.rb` 直接进行数据迁移：

```
redis@192.168.1.65:~> script/redis-trib.rb import --replace --from
192.168.1.7:6379 192.168.1.65:6379
...
Migrating key:974823 to 192.168.1.66:6380: OK
```

```
Migrating key:518820 to 192.168.1.66:6380: OK
Migrating key:704034 to 192.168.1.67:6381: OK
Migrating key:497427 to 192.168.1.65:6379: OK
Migrating key:349208 to 192.168.1.66:6380: OK
Migrating key:681758 to 192.168.1.67:6381: OK
Migrating key:190685 to 192.168.1.67:6381: OK
Migrating key:509235 to 192.168.1.66:6380: OK
Migrating key:635418 to 192.168.1.65:6379: OK
```

9.7.3 工作原理

在本案例中，我们首先介绍了在单个 Redis 实例间进行数据迁移的三种方法。

前两种方法利用了主从同步及加载持久化文件，这两种方法都很容易理解。

第三种方法使用了 `MIGRATE` 命令。正如 Redis 命令文档中所述，该命令实际上在源实例中执行了 `DUMP`，并在目的实例中执行了 `RESTORE`。`MIGRATE` 命令有两个选项，`COPY` 和 `REPLACE`，用于决定这个命令的行为。如果指定了 `COPY` 选项，那么源实例中的数据在迁移后不会被删除。此外，使用 `REPLACE` 选项可以在迁移过程中覆盖目的实例中已有的键。

之后，我们介绍了单实例与一个 Redis 集群间的数据迁移。前两种方法利用了 AOF 持久化作为数据迁移的媒介。第三种方法实际在内部使用了 `SCAN` 和 `MIGRATE` 命令从源实例向集群的一个节点复制键。通过计算键和槽的映射，`redis-trib.rb` 脚本会直接把键发送到相应的槽。所以，我们不需要重新定位键或进行槽分片。

9.7.4 更多细节

还有几个优秀的第三方工具可以用于 Redis 的数据迁移。其一是来自 CodisLabs 的 `redis-port` 工具，另一个是来自 VIPShop 的 `redis-migrate-tool` 工具。这两个项目都是开源的，可以从 GitHub 获取。由于本书篇幅的限制，我们不会详细讨论上述两个工具的细节。

9.7.5 相关内容

- 关于 Redis 集群数据迁移的完整参考手册，请参阅如下链接中的迁移到 Redis 集群一节：
<https://redis.io/topics/cluster-tutorial>。
- 有关 `MIGRATE` 命令的更多细节，请参阅：<https://redis.io/commands/migrate>。
- 有关 `redis-port` 项目的更多细节，请参阅：<https://github.com/CodisLabs/redis-port>。
- 有关 `redis-migrate-tool` 项目的更多细节，请参阅：
<https://github.com/vipshop/redis-migrate-tool>。

第 10 章

Redis 的故障诊断

在本章中，我们将学习下列案例：

- Redis 的健康检查。
- 使用 SLOWLOG 识别慢操作/查询。
- 延迟问题的故障诊断。
- 内存问题的故障诊断。
- 崩溃问题的故障诊断。

10.1 本章概要

我们在前几章中主要介绍了 Redis 的生产部署和日常管理任务。即使我们认真地进行了 Redis 实例的部署并完成了运维操作，但一旦 Redis 服务上线后，故障诊断仍然是在所难免的。在本章中，我们将详细介绍一些重要的 Redis 故障诊断方法。

首先，我们将学习日常的健康检查。日常的健康检查可能会在灾难发生之前向我们提前预警。之后，我们将介绍如何使用 SLOWLOG 命令来识别慢查询或慢操作。随后，我们将详细讨论如何处理延迟和内存问题。最后，我们将介绍 Redis 实例出现崩溃问题时的调试机制。

10.2 Redis 的健康检查

检查 Redis 服务器的健康状态对故障诊断来说非常重要。通过监控指标和统计信息，我们通常可以看到服务器上发生了什么、正在发生什么或者什么可能是问题出现的根源。在本案例中，我们

将介绍一些 Redis 服务器指标或统计数据。在对问题进行故障诊断时，我们应该留意这些指标或统计数据。

10.2.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

10.2.2 操作步骤

接下来，让我们按照以下的步骤来学习如何对 Redis 进行健康检查。

1. 使用 INFO 命令来获取不同维度的指标和统计数据：

```
127.0.0.1:6379> INFO STATS
# Stats
total_connections_received:1
total_commands_processed:18
instantaneous_ops_per_sec:0
total_net_input_bytes:671
total_net_output_bytes:13183
instantaneous_input_kbps:0.00
instantaneous_output_kbps:0.00
rejected_connections:0
sync_full:0
sync_partial_ok:0
sync_partial_err:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
...
latest_fork_usec:415
...
```

```
127.0.0.1:6379> INFO clients
# Clients
connected_clients:1
client_longest_output_list:0
```

```

client_biggest_input_buf:0
blocked_clients:0

127.0.0.1:6379> INFO persistence
# Persistence
loading:0
rdb_changes_since_last_save:0
rdb_bgsave_in_progress:0
rdb_last_save_time:1514662415
rdb_last_bgsave_status:ok
rdb_last_bgsave_time_sec:-1
rdb_current_bgsave_time_sec:-1
rdb_last_cow_size:0
aof_enabled:0
aof_rewrite_in_progress:0
aof_rewrite_scheduled:0
aof_last_rewrite_time_sec:-1
aof_current_rewrite_time_sec:-1
aof_last_bgrewrite_status:ok
aof_last_write_status:ok
aof_last_cow_size:0

```

2. 使用 redis-cli --stat 来持续地监控 Redis 的基本统计数据：

```

$ bin/redis-cli --stat
----- data ----- load ----- - child -
keys          mem          clients blocked requests          connections
680765        27.04G      541      0          60853686691 (+0)      2012384429 AO
680961        27.04G      541      0          60853687586 (+895)   2012384431 AOF
681162        27.04G      539      0          60853688521 (+935)   2012384433 AOF
681362        27.04G      539      0          60853689496 (+975)   2012384435 AOF
681549        27.04G      543      0          60853690312 (+816)   2012384442 AOF
...

```

10.2.3 工作原理

INFO STATS 命令返回 Redis 服务器的总体统计信息。我们应该注意的指标包括：

- `total_connections_received`: 服务器接受的总连接数。如果这个指标在短时间内快速增长, 那么 Redis 服务器可能会遇到 CPU 使用率过高的问题。
- `instantaneous_ops_per_sec`: 每秒处理的命令数。
- `rejected_connections`: 由于 `maxclients` 的限制而被拒绝的连接数。如果这个指标的值增加, 那么我们应该特别注意 Redis 的内存使用情况。
- `sync_full`: 从实例与该主实例完全同步的次数。
- `sync_partial_ok`: 完成部分同步的次数。
- `sync_partial_err`: 部分同步未能完成的次数。
- `evicted_keys`: 由于 `maxmemory` 的限制而被淘汰的键的数量。
- `keyspace_misses`: 在主词典中查找键失败的次数。如果这个指标太高, 那么我们应该考虑优化应用程序来降低未能命中键的查询的数量。
- `latest_fork_usec`: 上一次 `fork` 操作耗费的时间 (以微秒为单位)。

在 `Clients` 部分中, 我们应该监控以下指标:

- `client_longest_output_list`: 当前客户端连接中最长的输出列表。当这个指标的值超过 10 万时, 我们需要警惕。
- `client_biggest_input_buf`: 当前客户端连接中最大的输入缓冲区。当最大的输入缓冲区大小超过 10MB 时, 我们需要小心。
- `blocked_clients`: 被一个阻塞调用 (`BLPOP`、`BRPOP` 或 `BRPOPLPUSH`) 挂起的客户端数量。

在 `Persistence` 部分中, 我们应该特别注意的是 `rdb_last_cow_size` 和 `aof_last_cow_size`, 也就是最后一个 `BGSAVE` (保存 RDB 转储) 和 `BGREWRITEAOF` (AOF 重写) 操作使用的写时复制缓冲区的大小。

我们在第 9 章管理 Redis 中介绍了 Redis 的内存使用情况监控。我们绝对应该留意 Redis 的内存使用情况。人们经常忘记检查的另一件事是 Redis 数据目录的磁盘配额。如果磁盘空间已满, 那么 Redis 将不能保存 RDB 或 AOF 转储文件。

`redis-cli` 的 `--stat` 选项可以实时地输出基本的服务器指标, 这样我们就可以对内存使用情况、连接的客户端等有一个大致的了解。在默认情况下, 该命令每秒钟打印一行; 不过, 我们也可以使用 `-i <interval>` 选项指定间隔的时间。

10.2.4 相关内容

- `INFO` 命令的官方文档, 请参阅: <https://redis.io/commands/info>。

10.3 使用 SLOWLOG 识别慢查询

当 Redis 中存在性能问题时（例如查询操作经常超时），我们可能需要检查为什么 Redis 不能及时地完成查询请求。一个可能的原因是，有些查询或操作花费了很长的时间才完成。

在本案例中，我们将介绍如何使用 Redis 的 SLOWLOG 功能来识别慢查询或慢操作。

10.3.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤安装一个 Redis 服务器。

10.3.2 操作步骤

1. 出于演示的目的，让我们把 `slowlog-log-slower-than` 设为一个非常小的值：

```
127.0.0.1:6379> CONFIG SET slowlog-log-slower-than 5
```

2. 执行一些测试命令：

```
127.0.0.1:6379> SET foo bar
```

```
OK
```

```
127.0.0.1:6379> HGETALL bighash
```

```
...
```

```
127.0.0.1:6379> HMSET new_hash aaa bbb ccc ddd eee fff
```

```
OK
```

3. 我们可以使用 `SLOWLOG GET` 来读取所有的慢日志：

```
127.0.0.1:6379> SLOWLOG GET
```

```
1) 1) (integer) 6
```

```
2) (integer) 1514613453
```

```
3) (integer) 6
```

```
4) 1) "HMSET"
```

```
2) "new_hash"
```

```
3) "aaa"
```

```
4) "bbb"
```

```
5) "ccc"
```

```
6) "ddd"
```

```
7) "eee"
```

```

8) "fff"
5) "127.0.0.1:46142"
6) ""
2) 1) (integer) 5
   2) (integer) 1514613413
   3) (integer) 5
4) 1) "HGETALL"
   2) "bighash"
5) "127.0.0.1:46142"
6) ""

```

4. 我们可以使用 `SLOWLOG LEN` 来获取慢日志中记录的数量：

```

127.0.0.1:6379> SLOWLOG LEN
(integer) 2

```

5. 我们可以使用 `SLOWLOG RESET` 来清除所有记录：

```

127.0.0.1:6379> SLOWLOG RESET
OK
127.0.0.1:6379> SLOWLOG LEN
(integer) 0

```

10.3.3 工作原理

Redis 慢日志用于记录执行时间超过 `slowlog-log-slower-than` 所指定阈值（以微秒为单位）的查询或操作。

在前面的示例中，我们将阈值设置为 5 微秒，这意味着所有执行时间超过 5 微秒的查询或操作都会被慢日志记录下来。`slowlog-log-slower-than` 的默认值是 10000（10 毫秒）；当这个选项的值为负时表示禁用慢日志，当这个选项的值为 0 时表示记录所有的查询。

被记录的慢查询或慢操作被压入一个先进先出的队列中，其最大大小可以由 `slowlog-max-len` 指定（默认值为 128）。`SLOWLOG GET` 命令用于输出队列中的所有记录。或者，我们可以使用 `SLOWLOG GET N` 来获取最近的 N 条记录。

慢日志记录队列只在内存中，并且不会持久化到磁盘上，因此慢日志机制的速度很快。

在前面的示例中，有两条慢日志记录。每一条慢日志记录由六个字段组成（在 4.0 之前的版本中是四个字段）：

- 一条慢日志具有的唯一渐进标识符。

- 所记录命令被处理时的 unix 时间戳。
- 执行命令所需要的时间（以微秒为单位）。
- 命令参数组成的数组。
- 客户端 IP 地址和端口（仅 4.0 之后的版本支持）。
- 客户端名称，如果通过 `CLIENT SETNAME` 命令设置了客户端名称（仅 4.0 之后的版本支持）。

10.3.4 更多细节

值得注意的是，Redis 慢日志子系统只考虑了命令的实际执行时间，因为这个时间是服务器线程被阻塞且不能响应其他请求的时间。诸如磁盘 I/O 或网络传输的时间并不在考虑范围内。我们有可能在应用端发现延迟增加甚至超时，但在 Redis 中却找不到任何慢日志。在这种情况下，瓶颈不是慢查询或慢操作，而是诸如网络延迟增加等其他因素。

10.3.5 相关内容

- SLOWLOG 命令的官方文档，请参阅：<https://redis.io/commands/slowlog>。

10.4 延迟问题的故障诊断

设计 Redis 的一个重要目标之一是实现大量查询的高速处理。在大多数情况下，我们对 Redis 客户端和服务端之间的响应时间有严格的要求。因此，对于 Redis 在线服务来说，高延迟是最为致命的问题。在本案例中，我们将学习如何测量和发现 Redis 的延迟，并为读者提供一些可能的线索以定位延迟问题发生的原因。

10.4.1 准备工作

我们需要按照第 5 章复制（*Replication*）中配置 Redis 的复制机制的案例中所描述的步骤完成主从复制的配置。

出于演示的目的，我们导入大量的数据：

```
for i in `seq 10`  
do  
  nohup node generator.js hash 1000000 session:${i} &  
done
```

10.4.2 操作步骤

首先，进行基准延迟的测量。

1. 在将 Redis 服务上线前，先在 Redis 服务器所运行的主机上进行一次固有延迟测试 (intrinsic-latency):

```
$ bin/redis-cli --intrinsic-latency 60
Max latency so far: 1 microseconds.
Max latency so far: 8 microseconds.
Max latency so far: 21 microseconds.
Max latency so far: 29 microseconds.
Max latency so far: 38 microseconds.
Max latency so far: 39 microseconds.
Max latency so far: 40 microseconds.
Max latency so far: 44 microseconds.
Max latency so far: 56 microseconds.

1867256801 total runs (avg latency: 0.0321 microseconds / 32.13 nanoseconds
per run).
Worst run took 1743x longer than the average latency.
```

2. 在启动 Redis 服务之前，我们应该测量的另一种延迟是网络往返延迟。我们可以使用 redis-cli 工具的 --latency 选项获取网络往返延迟：

```
redis@192.168.1.68:~> bin/redis-cli -h 192.168.1.7 --latency
min: 0, max: 27, avg: 0.23 (5180 samples)
```

3. 在 Redis 服务上线后，我们可能需要判断 Redis 的延迟是否正常。这可以通过使用 redis-cli 的 --latency-history 和 -i 选项来监控 Redis 服务器的 PING 延迟实现。我们还可以进一步使用诸如 Logstash 或 Flume 之类的日志收集器来收集延迟日志并报警：

```
$ bin/redis-cli --latency-history -i 10 >>latency.log &
$ tail -n10 latency.log
0 1 0.14 334
0 1 0.15 335
0 1 0.15 336
0 1 0.15 337
0 1 0.14 338
0 1 0.14 339
```

```
0 1 0.15 340
0 1 0.15 341
0 1 0.15 342
```

在收到警报后，我们可以首先判断慢命令是否为延迟问题的根源。

1. 使用 `INFO COMMANDSTATS` 命令检查命令处理相关的统计信息：

```
$ bin/redis-cli INFO COMMANDSTATS
# Commandstats
cmdstat_hmset:calls=10000000,usec=1018137090,usec_per_call=101.81
cmdstat_select:calls=10,usec=7,usec_per_call=0.70
cmdstat_dbsize:calls=7,usec=8,usec_per_call=1.14
cmdstat_info:calls=9,usec=307,usec_per_call=34.11
cmdstat_monitor:calls=2,usec=4,usec_per_call=2.00
cmdstat_config:calls=6,usec=38,usec_per_call=6.33
cmdstat_slowlog:calls=2,usec=51,usec_per_call=25.50
cmdstat_command:calls=4,usec=1012,usec_per_call=253.00
```

2. 当 Redis 正在运行时，按照上一案例中所描述的步骤持续地监控 `SLOWLOG`：

```
127.0.0.1:6379> SLOWLOG GET 10
1) 1) (integer) 268
2) (integer) 1514429456
3) (integer) 12026
4) 1) "HMSET"
   2) "session:4e8c04b5-43bd-4d70-8071-82cae9792228"
   3) "field_0"
   4) "ullamco"
   5) "field_1"
   6) "anim"
5) "127.0.0.1:52644"
6) ""
...
```

除了慢命令之外，延迟还可能是由于 CPU 使用率过高引起的。

1. 检查 `redis-server` 进程的 CPU 使用情况：

```
$ ps aux | head -1; ps aux | grep redis-server
USER          PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
```



```
redis      8787 99.3  1.6 1104752 1069984 ?      RNs1 15:12  20:57 bin/redis-
server 0.0.0.0:6379
```

2. 检查 total_connections_received 指标:

```
$ bin/redis-cli INFO STATS |grep total_connections_received
total_connections_received:380
```

另一个可能导致延迟的因素是 Redis 服务器的持久化。

1. 检查最后一次 fork 耗费的时间:

```
$ bin/redis-cli INFO |grep "fork"
latest_fork_usec:332096
```

2. 检查指标 aof_delayed_fsync 是否正在增加, 并检查指标 aof_pending_bio_fsync 来判断是否由于 AOF 后台重写而存在挂起的 fsync 任务:

```
$ bin/redis-cli INFO |grep aof_delayed_fsync
aof_delayed_fsync:31
$ bin/redis-cli INFO |grep aof_pending_bio_fsync
aof_pending_bio_fsync:1
```

3. 搜索 Redis 运行日志来判断是否存在缓慢的 AOF fsync:

```
$ less log/redis.log
8787:M 29 Dec 11:41:54.942 * Asynchronous AOF fsync is taking too long (disk
is busy?). Writing the AOF buffer without waiting for fsync to complete,
this may slow down Redis.
$ sudo strace -p $(pidof redis-server) -f -T -tt -e fdatsync
[pid 8799] 11:42:00.549175 fdatsync(21 <unfinished ...>
[pid 12280] 11:42:00.578918 +++ exited with 0 +++
[pid 8787] 11:42:00.578941 --- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED,
si_pid=12280, si_uid=10086, si_status=0, si_utime=3454, si_stime=460} ---
[pid 8799] 11:42:00.631804 <... fdatsync resumed> ) = 0 <0.082599>
[pid 8787] 11:42:12.940408 --- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL}
---
[pid 8799] 11:42:12.941798 fdatsync(21) = 0 <0.023443>
[pid 8799] 11:42:13.042708 fdatsync(21) = 0 <0.065880>
[pid 8787] 11:42:16.827229 --- SIGALRM {si_signo=SIGALRM, si_code=SI_KERNEL}
---
```

```
[pid 8799] 11:42:16.831017 fdatsync(21) = 0 <0.117848>
[pid 8799] 11:42:17.074308 fdatsync(21) = 0 <0.030427>
...
$ iostat 1
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.25    7.13   27.27   15.69    0.00   49.66

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
nvme0n1             179.00         0.00         716.00         0         716
sda                 240.00         0.00       122212.00         0       122212
```

注意

strace 是 Linux 系统调用的跟踪器。strace 的性能开销相当大；因此，在生产环境中使用 strace 时应该非常谨慎。

我们还应该检查 Redis 实例是否使用了交换空间：

```
$ bin/redis-cli INFO|grep process_id
process_id:20116
$ awk '/VmSwap/{print $2 " " $3}' /proc/20116/status
0 kB
```

最后，使用 dstat 检查主机的网络利用率：

```
$ dstat -nt
-net/total- ----system----
recv  send|      time
  0      0 |29-12 09:10:10
74k 179k |29-12 09:10:11
105k 235k |29-12 09:10:12
91k  209k |29-12 09:10:13
106k 238k |29-12 09:10:14
```

10.4.3 工作原理

首先，我们对基准延迟进行了测量，包括系统的固有延迟和网络的往返延迟。系统的固有延迟是一个基线，我们可以通过它来确定系统的延迟程度（不包括 Redis 实例的延迟）。对于网络延迟，我们可以通过在客户端应用的主机上进行测试来模拟网络情况。

在我们的示例中，延迟的基线是 0.26 (0.03+0.23) ms（毫秒）。

之后，我们通过 `redis-cli` 将延迟指标收集到日志中。结果显示了最小（min）、最大（max）和平均延迟（average latency），以及 Redis 响应 PING 探测序列的时间。我们可以将日志的内容导入 Elasticsearch 中进行索引，供进一步的分析和报警。

如果捕捉到了运行缓慢的东西，我们可以按照上一节所提到的步骤来检测延迟发生的原因。

首先，我们使用 `INFO COMMANDSTATS` 命令检查命令处理的相关统计信息。该命令给我们提供了一个有关请求处理统计信息的大致情况。之后，我们使用 `SLOWLOG` 命令检查慢日志以定位究竟哪个命令处理得慢。如果读者在上述步骤中发现了问题，就需要考虑使用时间复杂度更低的操作来实现业务逻辑，或者使用诸如 `SCAN` 或 `DEL` 之类的非阻塞命令。

接下来，我们检查了 CPU 利用率。鉴于 Redis 的单线程模型，核心的 `redis-server` 进程只能使用处理器的一个核。因此，如果 `ps` 命令显示 `redis-server` 进程的 CPU 利用率接近 100%，那么就意味着 Redis 实例已经超负荷了。如果出现了 CPU 处理性能的瓶颈，那么我们应该注意应用程序中调用的 Redis 操作。诸如对大量数据进行排序等操作可能会导致 CPU 跑满，而大量客户端连接到 Redis 实例也可能导致相同的问题。我们随后检查的指标是客户端的总连接数。如果该指标在短时间内急剧增加，就说明一些客户可能没有正确地管理它们与 Redis 服务器之间的连接。频繁的连接和断开可能同时导致客户端和服务端端的延迟，也会导致 `redis-server` 进程的 CPU 使用率过高。在这种情况下，我们必须非常谨慎地管理应用程序与 Redis 服务器的连接。

之后，我们检查了持久化的开销是否是导致延迟问题的原因。Redis 的持久化机制可能导致较高的磁盘 I/O 延迟。

其中一种可能是 `fork` 引起的延迟。RDB 后台转储和 AOF 重写操作将创建新的进程并在主进程中导致延迟。保存了大数据集（大于 16GB）的 Redis 服务器可能会遇到较高的 `fork` 延迟。

另一种可能是高的 AOF 延时。在我们的示例中，我们可以从 `aof_delayed_fsync` 和 `aof_pending_bio_fsync` 指标中看出，Redis 服务器被缓慢的 `fsync` 调用拖慢了。此外，Redis 运行日志也说明磁盘被 AOF 的 `fsync` 操作拖慢了。另一种检测此类问题的方法是使用 `strace` 跟踪 `fdatsync` 函数。从操作系统的角度看，我们使用 `iostat` 命令发现了有一个 I/O 等待，而这正是磁盘 I/O 缓慢的信号。

在这种情况下，我们可以将配置选项 `no-appendfsync-on-rewrite` 设为 `yes` 来进行折衷。该选项保证重写正在进行时不会对 AOF 文件进行 `fsync` 操作。不过，不进行 `fsync` 就意味着在后台重写过程中，如果 `redis-server` 进程意外退出可能会有丢失数据的风险。如果这个选项没有太大帮助，那么可以考虑禁用 AOF 或将 `appendfsync` 设为 `none`。

网络拥塞是 Redis 延迟问题的另一个重要因素。我们可以使用 `dstat` 工具检查网络的利用率，来判断网络是否存在拥塞。

最后，我们检查了 `redis-server` 进程是否用到了交换空间。一个需要交换空间的进程意味着它在等待交换空间中的内存页从交换空间移动到内存时会被内核阻塞。

10.4.4 更多细节

如果我们已经检查了延迟问题所有可能的原因，但仍然没有线索，那么可以求助于内部的延迟检测。

1. 使用延迟看门狗（latency watchdog）查看延迟发生时调用栈的情况：

```
$ bin/redis-cli config set watchdog-period 500
OK
$ bin/redis-cli debug sleep 3
$ bin/redis-cli config set watchdog-period 0
OK
8787:signal-handler (1514447053)
--- WATCHDOG TIMER EXPIRED ---
EIP:
/lib/x86_64-linux-gnu/libc.so.6(+0x14e0eb) [0x7fe381eb40eb]
$ tail -n100 log/redis.log
...
Backtrace:
bin/redis-server 0.0.0.0:6379(logStackTrace+0x45) [0x46ab25]
bin/redis-server 0.0.0.0:6379(watchdogSignalHandler+0x1b) [0x46abfb]
/lib/x86_64-linux-gnu/libpthread.so.0(+0x11390) [0x7fe382141390]
/lib/x86_64-linux-gnu/libc.so.6(+0x14e0eb) [0x7fe381eb40eb]
bin/redis-server 0.0.0.0:6379(sdscatlen+0x59) [0x430869]
bin/redis-server 0.0.0.0:6379(feedAppendOnlyFile+0x373) [0x464863]
bin/redis-server 0.0.0.0:6379(propagate+0x5a) [0x42b86a]
bin/redis-server 0.0.0.0:6379(call+0x3c7) [0x42bd47]
bin/redis-server 0.0.0.0:6379(processCommand+0x3a7) [0x42c127]
bin/redis-server 0.0.0.0:6379(processInputBuffer+0x105) [0x43bc15]
bin/redis-server 0.0.0.0:6379(aeProcessEvents+0x13e) [0x42586e]
bin/redis-server 0.0.0.0:6379(aeMain+0x2b) [0x425c9b]
bin/redis-server 0.0.0.0:6379(main+0x4a6) [0x422866]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0) [0x7fe381d86830]
bin/redis-server 0.0.0.0:6379(_start+0x29) [0x422b69]
8787:signal-handler (1514447053) -----
...
```

在本例中，我们会发现函数 `feedAppendOnlyFile` 运行得缓慢。在 Redis 的源码中，我们可以找到这个函数。

2. 另一个我们可以使用的延迟检测框架是 `latency-monitor`：

```
127.0.0.1:6379> CONFIG SET latency-monitor-threshold 100
```

```
OK
```

```
127.0.0.1:6379> DEBUG SLEEP 1
```

```
OK
```

```
(1.00s)
```

```
127.0.0.1:6379> DEBUG SLEEP .25
```

```
OK
```

```
127.0.0.1:6379> LATENCY LATEST
```

```
1) 1) "command"
```

```
2) (integer) 1514451674
```

```
3) (integer) 250
```

```
4) (integer) 1000
```

```
127.0.0.1:6379> KEYS session:1:*
```

```
127.0.0.1:6379> LATENCY LATEST
```

```
1) 1) "command"
```

```
2) (integer) 1514510287
```

```
3) (integer) 2445
```

```
4) (integer) 26167
```

```
127.0.0.1:6379> LATENCY HISTORY command
```

```
1) 1) (integer) 1514451668
```

```
2) (integer) 1000
```

```
2) 1) (integer) 1514451674
```

```
2) (integer) 250
```

```
3) 1) (integer) 1514451721
```

```
2) (integer) 26167
```

在本例中，我们通过将延迟阈值设置为 100ms 启用了内部看门狗，休眠一段时间等待信息收集，然后关闭了看门狗。`LATENCY LATEST` 命令显示了延迟事件的名称、延迟事件发生时的 UNIX 时间戳、最新的一次的事件延迟（以毫秒为单位），以及该事件的历史最大延迟。`LATENCY HISTORY` 命令会返回正在跟踪的 160 个最新延迟事件。

最后要记住的是，由于某些不容易发现的硬件故障，服务器的内存可能已经变得很慢了。因此，使用硬件测试工具来测试内存的速度可能会给我们提供一些提示，告诉我们为什么 Redis 实例运行得缓慢。

10.4.5 相关内容

- 有关 Redis 延迟故障诊断的完整参考，请参阅官方文档：<https://redis.io/topics/latency>。
- 如果读者对 Redis 延迟监控框架感兴趣，请参阅官方文档：<https://redis.io/topics/latency-monitor>。
- 关于如何使用 sysbench 来测试内存的性能，请参阅博客：<http://tech.donghao.org/2016/11/30/using-sysbench-to-test-memory-performance/>。

10.5 内存问题的故障诊断

正如我们在本书开头所提到的，Redis 是一个基于内存的键值数据存储。因此，除了延迟问题之外，内存使用是另一种致命的问题。在本案例中，我们将学习如何针对 Redis 中的内存问题进行故障诊断。

10.5.1 准备工作

我们需要按照第 5 章复制（*Replication*）中配置 Redis 的复制机制的案例中所描述的步骤完成主从复制的配置。

10.5.2 操作步骤

当内存问题发生时，我们应该确保能够收到报警：

1. 首先，注意 `used_memory_human` 是否大于 `maxmemory_human`：

```
$ bin/redis-cli INFO MEMORY | egrep "used_memory_human|maxmemory_human"
used_memory_human:1016.45M
maxmemory_human:1.00G
```

2. 通过向 Redis 实例写入简单的键值对来定期地验证实例是否在正常工作：

```
127.0.0.1:6379> SET foo bar
(error) OOM command not allowed when used memory > 'maxmemory'.
```

3. 注意 `evicted_keys` 指标增长的情况：

```
$ bin/redis-cli INFO Stats | grep evicted_keys
evicted_keys:382901
```

4. 接下来，检查 `redis-server` 进程的系统内存和交换空间使用情况：


```
$ bin/redis-cli INFO MEMORY |grep used_memory_rss_human
used_memory_rss_human: 1.03G
$ bin/redis-cli INFO|grep process_id
process_id: 9909
$ awk '/VmSwap/{print $2 " " $3}' /proc/9909/status
0 kB
```

如果我们认为 Redis 实例可能存在内存问题，检查该问题的根本原因是大数据集导致的，还是不正常的内部内存使用导致的：

```
$ bin/redis-cli INFO MEMORY | egrep "used_memory_dataset|
    used_memory_dataset_perc"
used_memory_dataset:980896614
used_memory_dataset_perc:91.43%
```

执行以下的操作来检查数据集的内存使用情况：

1. 使用 redis-cli 的 --bigkeys 选项搜索 Redis 实例中的巨大键：

```
$ bin/redis-cli --bigkeys
# Scanning the entire keyspace to find biggest keys as well as
# average sizes per key type. You can use -i 0.1 to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).
[00.00%] Biggest hash found so far 'sessionB:1:a166b25c-6199-459f-843a-278
a02303d61' with 5 fields
[00.00%] Biggest hash found so far 'sessionB:5:ecd01063-d25b-4335-8ebc-3
fbbd5a63fae' with 6 fields
[00.00%] Biggest hash found so far 'sessionB:10:ede50aa3-ae13-4773-849c-22
a0a1992f6d' with 7 fields
[00.00%] Biggest hash found so far 'sessionB:8:87b7e610-d756-45cc-b52e-2
b19721adbc8' with 8 fields
[00.00%] Biggest hash found so far 'sessionB:3:5399b960-3f77-40db-b0e1-1
a66610ac669' with 9 fields
[00.01%] Biggest hash found so far 'sessionB:8:311037ce-a9ee-4a1c-b6c7-17
cce52ebb0b' with 10 fields
```

2. 另一种查找巨大键的方法是使用 SCAN 和 DEBUG OBJECT/MEMORY USAGE 命令遍历（译者注：请注意 DEBUG OBJECT 可能会运行缓慢）：

```
$ cat scanbigkeys.sh
```

```

#!/bin/bash
BEGIN=0
TOPNUM=10
TMPFILE="/tmp/scan.out"
RESULTFILE="result.out"
> $RESULTFILE
bin/redis-cli --raw SCAN $BEGIN > $TMPFILE
while true
do
    for key in `sed '1d' $TMPFILE`
    do
        echo -n $key" " >> $RESULTFILE
        bin/redis-cli --raw DEBUG OBJECT $key | awk '{print $NF}' | cut -d":" -f
            2 >> $RESULTFILE
    done

    CURSOR=`head -n1 $TMPFILE`

    if [[ $CURSOR -eq $BEGIN ]]
    then
        echo "Scan ended!"
        echo "The Top $TOPNUM key in your Redis is"
        cat $RESULTFILE | sort -nrk2 | head -n$TOPNUM
        exit
    fi

    bin/redis-cli --raw SCAN $CURSOR > $TMPFILE
done

$ bash scanbigkeys.sh
Scan ended!
The Top 10 key in your Redis is
sessionB:1:c8b22ce6-1cc0-4d8c-8d44-ac8b2d5856a5 268
sessionB:1:c7df3b53-359d-4d02-aca6-33695f7c4463 268
sessionB:1:643e949f-6c83-4112-a722-67e0735a205e 268
sessionB:1:9fe5d654-dc4d-45db-bd62-64c16cf4332f 266
sessionB:1:864aa1a1-5a99-45cb-88c2-0921c2aa6df3 265
sessionB:1:eea29894-5061-4e6e-be0d-11977bfc4615 264

```

非卖品！！严禁（售卖和上传互联网平台）！！
 仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

```
sessionB:1:71836c1c-032f-435d-901d-2ac976c9a354 263
sessionB:1:b9d8c5bf-e3cc-44f3-b3d9-d44152fb493f 262
sessionB:1:49a82365-5798-4726-aec4-d0136d4af422 262
sessionB:1:f966ef61-59fc-4d0f-adf2-05292524face 261
```

3. 或者，使用 Redis RDB Tools 生成内存使用情况的报告：

```
$ git clone https://github.com/sripathikrishnan/redis-rdb-tools
$ cd redis-rdb-tools
$ sudo python setup.py install
$ rdb -c memory dump.rdb --bytes 128 -f memory.csv
$ sort -t, -k4nr memory.csv | more
0,list,mylist,11149484,quicklist,2319,10000
0,hash,myset:000000020924,21036,hashtable,3,10000
0,hash,myset:000000000702,20844,hashtable,2,10000
...
```

执行以下的操作来检查内部的内存使用情况。

1. 启动 redis-benchmark 工具来模拟查询缓冲区导致的内存问题：

```
$ bin/redis-benchmark -q -d 102400000 -n 10000000 -r 25000 -t set
```

2. 检查内存相关的指标，并特别留意指标 `used_memory_human`、`used_memory_dataset`、`used_memory_overhead` 和 `used_memory_dataset_perc`：

```
$ bin/redis-cli INFO MEMORY
# Memory
used_memory:6005338152
used_memory_human:5.59G
used_memory_rss:6092353536
used_memory_rss_human:5.67G
used_memory_peak:6005338152
used_memory_peak_human:5.59G
used_memory_peak_perc:100.00%
used_memory_overhead:5122882822
used_memory_startup:765576
used_memory_dataset:882455330
used_memory_dataset_perc:14.70%
total_system_memory:67467202560
```



```
total_system_memory_human:62.83G
used_memory_lua:37888
used_memory_lua_human:37.00K
maxmemory:1073741824
maxmemory_human:1.00G
maxmemory_policy:noeviction
mem_fragmentation_ratio:1.01
mem_allocator:jemalloc-4.0.3
active_defrag_running:0
lazyfree_pending_objects:0
```

3. 检查客户端查询缓冲区的内存占用情况，并测试是否可以向 Redis 实例写入一个简单的键：

```
$ bin/redis-cli client list | awk 'BEGIN{sum=0} {sum+=substr($12,6);sum+=
    substr($13,11)}END{print sum}'
5120032868
$ bin/redis-cli set foo bar
(error) OOM command not allowed when used memory > 'maxmemory'.
```

4. 要模拟客户端输出缓冲区的内存问题，使用 **Ctrl+C** 停止 `redis-benchmark` 工具，并将 `maxmemory-policy` 更改为 `allkeys-lru`。

5. 使用 `monitor` 选项启动 `redis-cli`，将输出重定向到一个日志文件，然后再次启动 `redis-benchmark`：

```
$ bin/redis-cli config set maxmemory-policy allkeys-lru
OK
$ nohup bin/redis-cli monitor >monitor.log &
[1] 25778
$ for i in `seq 5`
do
    nohup bin/redis-benchmark -P 4 -t lpush -r 5000000 -n 10000000 -d 100 &
done
```

6. 等待一段时间，并使用以下的命令来获取内存相关的指标：

```
$ bin/redis-cli dbsize;bin/redis-cli INFO MEMORY; bin/redis-cli INFO CLIENTS
(integer) 0
# Memory
used_memory:1073795088
```

```

used_memory_human:1.00G
used_memory_rss:1253953536
used_memory_rss_human:1.17G
used_memory_peak:1073795088
used_memory_peak_human:1.00G
used_memory_peak_perc:100.00%
used_memory_overhead:13222706
used_memory_startup:765576
used_memory_dataset:1060572382
used_memory_dataset_perc:98.84%
total_system_memory:67467202560
total_system_memory_human:62.83G
used_memory_lua:37888
used_memory_lua_human:37.00K
maxmemory:1073741824
maxmemory_human:1.00G
maxmemory_policy:allkeys-lru
mem_fragmentation_ratio:1.17
mem_allocator:jemalloc-4.0.3
active_defrag_running:0
lazyfree_pending_objects:0
# Clients
connected_clients:252
client_longest_output_list:32248
client_biggest_input_buf:0
blocked_clients:0

```

7. 获取输出缓冲区大小排前 10 的客户端，然后测试我们是否仍然能够向 Redis 实例写入一个简单的键：

```

$ bin/redis-cli CLIENT LIST | awk '{print substr($16,6), $1,$16,$18}' | sort -
nrk1,1 | cut -f1 -d" " --complement | head -n10
id=62 omem=527221432 cmd=monitor
id=999 omem=0 cmd=lpush
id=998 omem=0 cmd=lpush
id=997 omem=0 cmd=lpush
id=996 omem=0 cmd=lpush
id=995 omem=0 cmd=lpush

```

```

id=994 omem=0 cmd=lpush
id=993 omem=0 cmd=lpush
id=992 omem=0 cmd=lpush
id=991 omem=0 cmd=lpush
127.0.0.1:6379> set foo bar
(error) OOM command not allowed when used memory > 'maxmemory'.

```

8. 要判断是否是内存碎片开销导致的问题，可以检查内存碎片率是否大于 1.5:

```

$ bin/redis-cli INFO MEMORY|grep mem_fragmentation_ratio
mem_fragmentation_ratio:1.03

```

10.5.3 工作原理

在深入探讨如何对内存问题进行故障诊断之前，我们首先需要关注 Redis 是如何计算内存并检查存在内存不足（OOM，out-of-memory）问题的。

一般来说，一个 Redis 实例的内存结构如图 10-1 所示。

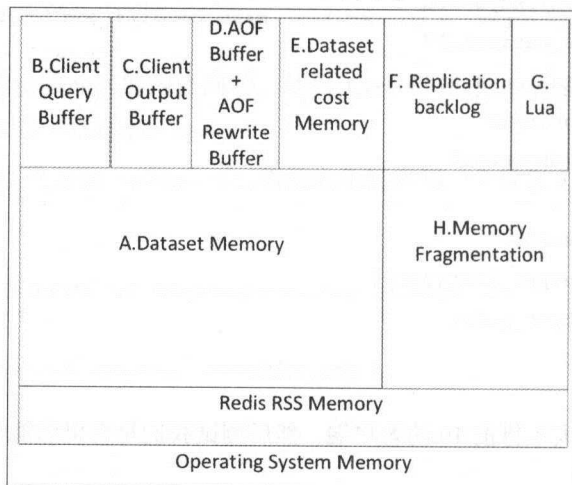


图 10.1 内存结构

由于我们已经在前几章的案例中对上图每个部分的含义进行了学习，所以在此不再展开。对于 Redis 4.0.1 来说，我们需要尤为关注的一部分内存是客户端查询缓冲区。虽然指标 `used_memory` 包括了这一部分内存，但它会随客户端查询缓冲区的增长而扩展。因此，如果查询缓冲区异常增长，我们可能会发现 `used_memory` 会比 `maxmemory` 大得多。

在这种情况下，如果配置了正确的内存策略，那么 Redis 将开始淘汰键。另一件值得一提的事是，客户端输出缓冲区不包含在 `INFO MEMORY` 命令返回的 `used_memory_overhead` 指标中，但包

含在 `used_memory` 指标中。因此，即使 Redis 实例中没有键，如果客户端输出缓冲区被滥用那么 OOM 问题仍然可能发生。

请记住以下两个公式。第一个公式说明了 Redis 如何判断是否可能发生 OOM 问题：

```
used_memory - AOF_buffer_size - slave_output_buffer_size >= maxmemory
```

第二个公式说明了 `used_memory_overhead` 是如何计算的：

```
used_memory_overhead = server_initial_memory_usage + repl_backlog +
    slave_clients_output_buffer + normal_clients_output_buffer +
    pubsub_clients_output_buffer + normal_clients_query_buffer +
    clients_metadata + AOF buffer + AOF rewrite buffer
```

在 Redis 中，OOM 问题是最常见的问题。正如我们在之前的案例中所描述的，一旦发生 OOM 问题，Redis 就会试图根据淘汰策略来淘汰某些键。如果淘汰策略是 `noeviction`，那么新传入的写入操作会被拒绝且客户端将得到 OOM 的错误消息。

在上一节中，我们提到的第一件事是当内存问题发生时如何得到报警。我们有四种方法来监控异常的内存使用。在前两种方法中，我们试图确认 Redis 实例存储的数据集大小是否超过了配置的 `maxmemory` 选项。在后两种方法中，我们检查了 `redis-server` 进程的驻留集大小（**Resident Set Size, RSS**），来判断是否占用了太多的内存而导致操作系统开始使用交换空间，或者甚至打算启动 Linux OOM Killer 来终止进程。

在我们确认了内存问题发生之后，接下来要做的就是找出导致问题的根本原因，不管原因是大数据集还是不正常的内部内存使用。通过检查指标 `used_memory_dataset` 和 `used_memory_dataset_perc`，我们可以判断导致内存问题的根本原因是大数据集（如果 `used_memory_dataset_perc` 大于 90%）还是不正常的内部内存使用。

排除大数据集问题的主要方法是定位 Redis 实例中的巨大键。执行搜索最简单的方法是使用 `redis-cli` 工具的 `--bigkeys` 选项。另一种方法则是利用 `SCAN` 和 `DEBUG OBJECT` 命令。另外一种分析数据集的方法是使用第三方的 Redis RDB Tools，该工具通过解析 RDB 文件来生成一个内存使用报告。

Redis 服务器的内部内存开销比较复杂。我们首先检查了开销是否是客户端查询缓冲区导致的。出于演示的目的，我们启动了 `redis-benchmark` 工具来导入数据，并将每个键的有效负载设为 100MB（这么大的负载会导致客户端查询缓冲区快速增长）。我们注意到，指标 `used_memory_human` 会比指标 `maxmemory` 大得多，而指标 `used_memory_dataset_perc` 还不到 20%。接下来，我们将所有客户端的查询缓冲区相加，并发现它们占用了大约 4.6GB（512,0032,868 字节）的内存。此时，由于 OOM 问题，我们已经无法将一个简单的键值对写入到 Redis 实例中。在这种情况下，我们建议停止应用程序或终止连接。

接下来，我们展示了监控客户端输出缓冲区的内存问题。出于监控的目的，我们有时会使用 `monitor` 选项启动 `redis-cli` 工具。有时，在完成监控数据的收集后，我们可能会忘记停止监控进程。我们通过将 `redis-cli monitor` 的输出重定向到一个日志文件来模拟了这种情况。在繁重的流量下，监控客户端的输出缓冲区可能会占用大量内存。

在我们的示例中，即使 Redis 实例中没有键（由于内存限制和淘汰策略，测试中唯一的键也被淘汰了），但是 OOM 问题仍旧会发生。通过检查客户端列表，我们可以发现问题的根源在于监控输出缓冲区的内存开销。应该注意，这种开销并不包含在指标 `used_memory_overhead` 中。要解决这个问题，只需停止或杀死监控客户端即可。

最后，由于高度频繁地写入和删除键，Redis 可能会在其 RSS 内存中留下一些内存碎片。而跟踪碎片率也是很重要的，如果发现内存碎片率大于 1.5，那么我们可以使用 `MEMORY PURGE` 命令释放内存碎片。如果这没有太大的帮助，则必须在 Redis 实例上进行一次故障转移，并计划一次重新启动以完全释放内存碎片。

10.5.4 更多细节

在上一节中，我们提到，在本书编写时，对于 Redis 4.0.1 而言，指标 `used_memory_overhead` 并不包括监控客户端输出缓冲区。有一个代码提交（PR，pull request）解决了这个问题，参见 Redis 在 GitHub 代码仓库中的 Issue #4502。

10.5.5 相关内容

- 我们可以通过做很多工作来进行 Redis 的内存优化。读者可以参考 Redis 作者提供的如下文档：<https://redis.io/topics/memory-optimization>。
- 有关内存优化的多种技巧，请参阅：<https://github.com/sripathikrishnan/redis-rdb-tools/wiki/Redis-Memory-Optimization>。
- 对于 Redis RDB Tools，请参阅：<https://github.com/sripathikrishnan/redis-rdb-tools>。
- 另一个流行的内存分析工具 Redis Memory Analyzer，请参阅：<https://github.com/gamenet/redis-memory-analyzer>。

10.6 崩溃问题的故障诊断

在极少数情况下，Redis 在遇到 Bug 时可能会崩溃。当 Redis 实例崩溃时，我们可以进行很多操作。在本案例中，我们将学习如何在 Redis 中对崩溃问题进行故障诊断。

10.6.1 准备工作

我们需要按照第 5 章复制（*Replication*）中配置 *Redis* 的复制机制的案例中所描述的步骤完成主从复制的配置。

10.6.2 操作步骤

接下来，让我们按照如下的步骤来学习如何进行故障诊断。

1. 使用 `DEBUG SEGFAULT` 命令模拟 *Redis* 实例的崩溃：

```
$ bin/redis-cli DEBUG SEGFAULT
Error: Server closed the connection
```

2. 在 *Redis* 崩溃后，检查 *Redis* 实例的日志：

```
=== REDIS BUG REPORT START: Cut & paste starting from here ===
1710:M 07 Jan 20:58:01.324 # Redis 4.0.1 crashed by signal: 11
1710:M 07 Jan 20:58:01.324 # Crashed running the instruction at: 0x469cd0
1710:M 07 Jan 20:58:01.324 # Accessing address: 0xffffffffffffffff
1710:M 07 Jan 20:58:01.324 # Failed assertion: <no assertion failed> (<no file
>:0)

----- STACK TRACE -----
EIP:
bin/redis-server 0.0.0.0:6379(debugCommand+0x250) [0x469cd0]
Backtrace:
bin/redis-server 0.0.0.0:6379(logStackTrace+0x45) [0x46ab25]
...
bin/redis-server 0.0.0.0:6379(_start+0x29) [0x422b69]
----- INFO OUTPUT -----
# Server
redis_version:4.0.1
redis_git_sha1:00000000
redis_git_dirty:0
...
# Keyspace
----- CLIENT LIST OUTPUT -----
...
----- CURRENT CLIENT INFO -----
....
```



```
----- REGISTERS -----
1710:M 07 Jan 20:58:01.344 #
RAX:0000000000000000 RBX:00007f16e9aaaa40
...
----- DUMPING CODE AROUND EIP -----
Symbol: debugCommand (base: 0x469a80)
...
=== REDIS BUG REPORT END. Make sure to include from START to END. ===

Please report the crash by opening an issue on github:
http://github.com/antirez/redis/issues
Suspect RAM error? Use redis-server --test-memory to verify it.
```

3. 如果能够复现崩溃的情况，那么在测试环境中使用 **GDB**（GNU 项目调试器，GNU Project Debugger）来调试 redis-server 进程，有助于了解 Redis 实例崩溃时发生的情形：

首先，我们获取 redis-server 的进程 ID：

```
$ bin/redis-cli info | grep process_id
process_id: 11239
```

4. 打开另一个终端，然后使用 gdb 附加到进程：

```
$ gdb /redis/bin/redis-server 11239
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
...
Reading symbols from /redis/bin/redis-server...done.
Attaching to program: /redis/bin/redis-server, process 9611
[New LWP 9612]
[New LWP 9613]
[New LWP 9614]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007f0eddeed9d3 in epoll_wait () at ../sysdeps/unix/syscall-template.S:84
84 ../sysdeps/unix/syscall-template.S: No such file or directory.
```

5. 让进程继续运行：

```
(gdb) continue
Continuing.
```

6. 让 Redis 实例崩溃，GDB 会打印出段错误的日志：

```
$ bin/redis-cli DEBUG SEGFAULT
(gdb) c
Continuing.
Thread 1 "redis-server" received signal SIGSEGV, Segmentation fault.
debugCommand (c=0x7f9e0b1b4200) at debug.c:313
313             *((char*)-1) = 'x';
```

7. 获取栈回溯和处理器的寄存器信息：

```
(gdb) bt
#0  debugCommand (c=0x7f9e0b1b4200) at debug.c:313
#1  0x000000000042ba26 in call (c=c@entry=0x7f9e0b1b4200, flags=flags@entry
    =15) at server.c:2199
#2  0x000000000042c127 in processCommand (c=0x7f9e0b1b4200) at server.c:2479
...
#5  0x0000000000425c9b in aeMain (eventLoop=0x7f9e14a36050) at ae.c:464
#6  0x0000000000422866 in main (argc=<optimized out>, argv=0x7ffd03cc8978) at
    server.c:3844
(gdb) info registers
rax                0x0            0
rbx                0x7f9e0b1b4200    140316767896064
...
gs                 0x0            0
```

8. 检查变量的值：

```
(gdb) p /s (char*)c->argv[0]->ptr
$15 = 0x7f9e0b756e13 "DEBUG"
(gdb) p /s (char*)c->argv[1]->ptr
$16 = 0x7f9e0b756e33 "SEGFAULT"
(gdb) p server
$1 = {pid = 11239, configfile = 0x7f9e14a1c093 "/redis/conf/redis.conf",
    executable = 0x7f9e14a2f003 "/redis/bin/redis-server",
    exec_argv = 0x7f9e14a21cd0, hz = 10, db = 0x7f9e14b4a000, commands = 0
    x7f9e14a18060, orig_commands = 0x7f9e14a180c0,
    el = 0x7f9e14a36050, lruclock = 5383977, shutdown_asap = 0, activerehashing =
    1, active_defrag_running = 0, requirepass = 0x0,
```

```
latency_events = 0x7f9e14a1a280, assert_failed = 0x506e30 "<no assertion
failed>", assert_file = 0x506e46 "<no file>",
assert_line = 0, bug_report_start = 0, watchdog_period = 0, system_memory_size
= 67467202560, lruclock_mutex = {__data = {
__lock = 0, __count = 0, __owner = 0, __nusers = 0, __kind = 0, __spins
= 0, __elision = 0, __list = {__prev = 0x0,
__next = 0x0}}, __size = '\000' <repeats 39 times>, __align = 0},
next_client_id_mutex = {__data = {__lock = 0,
__count = 0, __owner = 0, __nusers = 0, __kind = 0, __spins = 0,
__elision = 0, __list = {__prev = 0x0, __next = 0x0}},
__size = '\000' <repeats 39 times>, __align = 0}, unixtime_mutex = {__data
= {__lock = 0, __count = 0, __owner = 0,
__nusers = 0, __kind = 0, __spins = 0, __elision = 0, __list = {__prev =
0x0, __next = 0x0}},
__size = '\000' <repeats 39 times>, __align = 0}}
```

9. 生成 core 文件:

```
(gdb) gc core
Saved corefile core.11239
```

10. 最后, 退出 gdb:

```
(gdb) q
A debugging session is active.
Inferior 1 [process 11239] will be detached.
Quit anyway? (y or n) y
Detaching from program: /redis/bin/redis-server, process 11239
```

我们还可以通过邮件将日志和 core 文件发送给 Redis 的作者 (antirez@gmail.com)。

10.6.3 工作原理

在第 1 步中, 我们使用 `DEBUG SEGFAULT` 命令在 Redis 实例中触发了一个段错误。实例立即就崩溃了。从 Redis 实例的运行日志中可以清楚地看到进程已经崩溃, 且所有的信息 (包括栈回溯、`INFO` 命令的输出、客户端信息及寄存器) 都被记录到了日志文件中。如果崩溃可以复现, 那么我们可以使用 GDB 附加到 Redis 进程上进行进一步的调试。请注意, 我们应该总是在测试环境中进行测试。在将 GDB 附加到 `redis-server` 进程之后, 可以使用 `c` 命令让 Redis 进程继续运行。出于演

示的目的，我们再次使用 `DEBUG SEGFAULT` 命令使 Redis 实例崩溃。GDB 捕获到了段错误并打印出了消息。随后，我们使用 GDB 的 `bt` 命令检查了栈回溯，并使用 `info registers` 命令查看了寄存器信息。通过检查 Redis 的源代码，我们还可以使用 GDB 的 `p` 命令获取重要变量的值（译者注：如果编译 Redis 时保留了调试信息）。最后，我们生成了一个可用于后续调试的 `core` 文件并使用 `q` 命令退出了 GDB。

限于本书的篇幅，我们不会深入介绍调试的细节。在这种情况下，将 `core` 文件和其他信息发送给 Redis 的作者是一个不错的主意。如果我们认为发现了一个真正的问题，也可以在 GitHub 上创建一个 Issue。

10.6.4 相关内容

- 有关 GDB 的更多细节，请参阅：<https://www.gnu.org/s/gdb/documentation>。
- 有关 Redis 调试的更多细节，请参阅：<https://redis.io/topics/debugging>。
- Redis 的作者在他的博客上谈到了有关 Redis 调试的很多内容，请参阅：<http://antirez.com/news/43>。

第 11 章

使用 Redis 模块扩展 Redis

在本章中，我们将学习下列案例：

- 加载 Redis 模块。
- 编写 Redis 模块。

11.1 本章概要

尽管 Redis 支持大量的数据类型和优秀特性，但有时我们可能会希望将自定义的数据类型或命令添加到 Redis 中。在第 3 章数据特性中，我们已经了解到 Lua 脚本可用于在 Redis 内置数据类型和命令之上实现定制的逻辑。此外，从 Redis 4.0 开始，我们还可以使用 Redis 模块来扩展 Redis 的功能。

Redis 模块是共享的 C 语言库，可以在启动或运行时由 Redis 服务器加载。与 Lua 脚本相比，Redis 模块具有以下优势：

- 作为 C 语言库，Redis 模块的运行速度要比 Lua 脚本快得多
- 在 Redis 模块中我们可以创建新的数据结构，而在 Lua 脚本中只能使用现有的数据类型
- 像原生命令一样，在 Redis 模块中创建的命令可以直接从客户端调用；而 Lua 脚本必须使用 EVAL 或 EVALSHA 命令调用
- 可以将第三方库链接到 Redis 模块，而在 Lua 脚本中所能做的事情是非常有限的
- 在 Redis 模块中可以使用的 API 比暴露给 Lua 脚本的 API 要丰富得多

在本章中，我们将首先介绍如何加载现有的 Redis 模块。然后，我们将学习如何使用 Redis 的模块 API 编写一个简单的 Redis 模块。

11.2 加载 Redis 模块

我们可以使用很多开源的 Redis 模块来扩展 Redis 的功能。在本案例中，我们将学习如何在 Redis 服务器上加载模块。我们将加载 ReJSON 模块，该模块用于向 Redis 中增加 JSON 数据类型的支持。

11.2.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤完成 Redis 服务器的安装。

11.2.2 操作步骤

接下来，让我们按照以下的步骤来学习如何加载 Redis 模块：

1. 从 GitHub 下载 ReJSON 的源代码：

```
$ git clone https://github.com/RedisLabsModules/rejson.git
Cloning into 'rejson'...
remote: Counting objects: 2066, done.
remote: Compressing objects: 100% (64/64), done.
remote: Total 2066 (delta 52), reused 83 (delta 43), pack-reused
1952
Receiving objects: 100% (2066/2066), 3.68 MiB | 0 bytes/s, done.
Resolving deltas: 100% (1114/1114), done.
Checking connectivity... done.
```

2. 切换到模块的源码目录，然后运行 make 编译模块：

```
$ cd rejson/src
~/rejson/src$ make
```

编译生成的二进制文件为 rejson.so：

```
~/rejson/src$ ls -l rejson.so
-rwxrwxr-x 1 user user 455888 Dec 29 17:56 rejson.so
```

3. 要加载模块，我们可以在 redis-cli 中执行 MODULE LOAD 命令，该命令后面接的是模块二进制可执行文件的路径：

```
127.0.0.1:6379> MODULE LOAD /redis/rejson/src/rejson.so
OK
```


4. 测试模块是否已被加载：

```
127.0.0.1:6379> JSON.SET jsonKey . '{"foo": "bar", "baz": ["aaa", "bbb"]}'
OK
127.0.0.1:6379> JSON.GET jsonKey
{"\"foo\": \"bar\", \"baz\": [\"aaa\", \"bbb\"]}"
```

5. 我们可以使用 MODULE UNLOAD 命令外加模块名称来卸载一个模块：

```
127.0.0.1:6379> MODULE UNLOAD ReJSON
(error) ERR Error unloading module: the module exports one or more module-side
data types, can't unload
```

6. 我们可以使用 MODULE LIST 命令来列出所有的已加载模块：

```
127.0.0.1:6379> MODULE LIST
1) 1) "name"
   2) "ReJSON"
   3) "ver"
   4) (integer) 10001
```

7. 我们还可以在配置文件中通过 loadmodule 指令加载模块：

```
loadmodule /redis/rejson/src/rejson.so
```

11.2.3 工作原理

加载 Redis 模块非常简单；当一个模块成功加载后，我们还可以从 Redis 服务器的日志中看到：

```
1960:M 29 Dec 18:31:07.403 # <ReJSON> JSON data type for Redis v1.0.1 [encver
0]
1960:M 29 Dec 18:31:07.403 * Module 'ReJSON' loaded from /redis/rejson/src/
rejson.so
```

MODULE UNLOAD 命令的参数是模块名而不是模块库文件的路径，因此保持模块库的文件名与模块名称一致是一个良好的实践。

在前面的示例中，由于 ReJSON 模块创建了一种新的数据类型，所以它不能在运行时卸载。如果模块是通过 MODULE LOAD 命令加载的，那么当服务器停止时会被自动卸载。

11.2.4 相关内容

- 以下链接给出了我们可以加载并使用的 Redis 模块列表：<https://redis.io/modules>。
- Redis 模块中心，请参考：<http://redismodules.com/>。
- 有关 Redis 模块的详细信息，请参阅 Redis 模块的介绍：<https://redis.io/topics/modules-intro>。

11.3 编写 Redis 模块

Redis 为需要编写模块的用户提供了大量的 C 语言 API。虽然 Redis 模块可以使用任何具有 C 语言绑定功能（C binding）的语言编写，但使用 C 语言编写 Redis 模块是最简单和最直接的方法。

在本案例中，我们将使用 C 语言实现一个名为 MYMODULE 的模块，该模块中提供了一个新的命令 ZIP，其调用方式如下：

```
MYMODULE.ZIP destination_hash field_list value_list
```

该命令以两个 Redis 列表为参数，并尝试创建一个哈希。其中，哈希的字段是 field_list 中的元素，而对应的值是 value_list 中位于相同索引处的元素。

例如，如果 field_list 为 ["john", "alex", "tom"], value_list 为 ["20", "30", "40"]，那么 destination_hash 将是 {"john": "20", "alex": "30", "tom": "40"}。如果两个列表的长度不相等，那么在创建哈希前较长的列表将被截断至较短列表的长度（在此之后列表本身不会改变）。如果 field_list 中有重复元素，那么位于最右索引处的值将保留在哈希中。该命令的返回值为所创建哈希的长度。

11.3.1 准备工作

我们需要按照第 1 章开始使用 Redis 中下载和安装 Redis 的案例所描述的步骤完成 Redis 服务器的安装。

11.3.2 操作步骤

编写 Redis 模块 MYMODULE 的步骤如下：

1. 创建一个文件 mymodule.c，然后在其中包含头文件 redismodule.h：

```
#include "redismodule.h"
```

2. 创建函数 RedisModule_OnLoad()：

```

int RedisModule_OnLoad(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {
    if (RedisModule_Init(ctx, "mymodule", 1, REDISMODULE_APIVER_1) ==
        REDISMODULE_ERR) {
        return REDISMODULE_ERR;
    }

    if (RedisModule_CreateCommand(ctx, "mymodule.zip", MyModule_Zip, "write
        deny-oom no-cluster", 1, 1, 1) == REDISMODULE_ERR) {
        return REDISMODULE_ERR;
    }

    return REDISMODULE_OK;
}

```

3. 创建函数 MyModule_Zip(), 也就是我们新命令的处理函数:

```

int MyModule_Zip(RedisModuleCtx *ctx, RedisModuleString **argv, int argc) {
    // MYMODULE.ZIP destination field_list value_list
    if (argc != 4) {
        return RedisModule_WrongArity(ctx);
    }

    RedisModule_AutoMemory(ctx);

    // Open field/value list keys
    RedisModuleKey *fieldListKey = RedisModule_OpenKey(ctx, argv[2],
        REDISMODULE_READ | REDISMODULE_WRITE);
    RedisModuleKey *valueListKey = RedisModule_OpenKey(ctx, argv[3],
        REDISMODULE_READ | REDISMODULE_WRITE);
    if ((RedisModule_KeyType(fieldListKey) != REDISMODULE_KEYTYPE_LIST &&
        RedisModule_KeyType(fieldListKey) != REDISMODULE_KEYTYPE_EMPTY) ||
        (RedisModule_KeyType(valueListKey) != REDISMODULE_KEYTYPE_LIST &&
        RedisModule_KeyType(valueListKey) != REDISMODULE_KEYTYPE_EMPTY)) {
        return RedisModule_ReplyWithError(ctx, REDISMODULE_ERRORMSG_WRONGTYPE);
    }

    // Open destination key
    RedisModuleKey *destinationKey = RedisModule_OpenKey(ctx, argv[1],

```

非卖品！！严禁（售卖和上传互联网平台）！！
 仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！


```

    REDISMODULE_WRITE);
    RedisModule_DeleteKey(destinationKey);

    //Get length of lists
    size_t fieldListLen = RedisModule_ValueLength(fieldListKey);
    size_t valueListLen = RedisModule_ValueLength(valueListKey);

    if (fieldListLen == 0 || valueListLen == 0) {
        RedisModule_ReplyWithLongLong(ctx, 0L);
        return REDISMODULE_OK;
    }

    size_t fCount = 0;
    size_t vCount = 0;
    while (fCount < fieldListLen && vCount < valueListLen) {
        //Pop from left and push back to right
        RedisModuleString *key = ListLPopRPush(fieldListKey);
        RedisModuleString *value = ListLPopRPush(valueListKey);

        //Set hash
        RedisModule_HashSet(destinationKey, REDISMODULE_HASH_NONE, key, value,
            NULL);
        fCount++;
        vCount++;
    }

    while (fCount++ < fieldListLen) {
        ListLPopRPush(fieldListKey);
    }

    while (vCount++ < valueListLen) {
        ListLPopRPush(valueListKey);
    }

    //Get hash length
    RedisModuleCallReply *hlenReply = RedisModule_Call(ctx, "HLEN", "s", argv
        [1]);

```

```

    if (hlenReply == NULL) {
        return RedisModule_ReplyWithError(ctx, "Failed to call HLEN");
    } else if (RedisModule_CallReplyType(hlenReply) == REDISMODULE_REPLY_ERROR)
    {
        RedisModule_ReplyWithCallReply(ctx, hlenReply);
        return REDISMODULE_ERR;
    }

    RedisModule_ReplyWithLongLong(ctx, RedisModule_CallReplyInteger(hlenReply));

    ;

    RedisModule_ReplicateVerbatim(ctx);
    return REDISMODULE_OK;
}

```

4. 创建函数 `ListLPopRPush()` 从列表中读取元素。请注意, `ListLPopRPush()` 和 `MyModule_Zip()` 应该放在 `RedisModule_OnLoad()` 之前, 否则就必须在文件的开头对它们进行声明:

```

static RedisModuleString *ListLPopRPush(RedisModuleKey *ListKey) {
    RedisModuleString *value = RedisModule_ListPop(ListKey,
        REDISMODULE_LIST_HEAD);
    RedisModule_ListPush(ListKey, REDISMODULE_LIST_TAIL, value);
    return value;
}

```

5. 从 Redis 的源码包中将 `redismodule.h` 拷贝到与 `mymodule.c` 的相同目录, 然后使用 `gcc` 进行编译:

```
gcc -fPIC -shared -std=gnu99 -o mymodule.so mymodule.c
```

6. 将生成的模块库文件复制到一个方便的位置 (例如 `/redis`), 然后在 Redis 中加载:

```

127.0.0.1:6379> MODULE LOAD /redis/mymodule.so
OK

```

7. 试用新命令:

```

127.0.0.1:6379> RPUSH fields john alex tom
(integer) 3
127.0.0.1:6379> RPUSH values 20 30 40
(integer) 4

```

```
127.0.0.1:6379> MYMODULE.ZIP myhash fields values
```

```
(integer) 3
```

```
127.0.0.1:6379> HGETALL myhash
```

```
1) "john"
```

```
2) "20"
```

```
3) "alex"
```

```
4) "30"
```

```
5) "tom"
```

```
6) "40"
```

11.3.3 工作原理

当模块在 Redis 中被加载时，将调用函数 `RedisModule_OnLoad()`。该函数是模块的入口点，在所有 Redis 模块中都必须实现。通常，我们需要先使用 `RedisModule_Init()` 初始化模块，并指定模块的名称、版本和 API 版本。Redis 模块被设计为与 Redis 服务器的版本不相关，但它们必须与 API 版本相兼容。也就是说，只要服务器知道模块使用的是哪个 API 版本，那么不需要重新编译模块也能将其加载到不同版本的 Redis 服务器中。目前 Redis 模块 API 版本号是 1 (`REDISMODULE_APIVER_1`)。我们还需要通过函数 `RedisModule_CreateCommand()` 给模块添加命令，该函数的参数是命令的名称、指向命令处理函数的指针和说明命令行为的一个字符串类型的标志。在 `MYMODULE.ZIP` 命令中，命令处理函数是在函数 `MyModule_Zip()` 中实现的。`MYMODULE.ZIP` 命令可能会修改 Redis 数据集，并且可能申请内存空间从而导致 **OOM (Out Of Memory)**；另外，`MYMODULE.ZIP` 命令不支持 Redis 集群。我们可以在 API 参考手册中找到完整的标志列表。`RedisModule_CreateCommand()` 函数的最后三个参数分别是第一个键的索引、最后一个键的索引和命令的键步长 (**key step**)。由于我们的命令只有一个键（对于目标哈希），我们只需传递 `(1, 1, 1)` 作为参数。如果要创建诸如 `MSET (MSET key value [key value]...)` 的命令，由于 `MSET` 可以传入无数个键（最后一个键的索引是 -1），则应该使用 `(1, -1, 1)`。除非有错误发生，否则 `RedisModule_OnLoad()` 函数应该返回 `REDISMODULE_OK`。

命令处理函数 `MyModule_Zip()` 包含三个参数：模块上下文、命令参数数组和命令参数个数。读者可能会注意到，几乎所有的 Redis 模块 API 都将模块上下文作为第一个参数。上下文用于获取对模块本身的引用，比如模块支持的命令、调用命令的客户端等等。该上下文将被其他模块 API 传入，而参数数组将由调用命令的客户端传入。

在函数内部，我们首先检查命令参数的个数是否正确。由于命令有三个参数 (`destination_hash`、`field_list` 和 `value_list`)，再加上命令本身，参数的个数应该是 4 个。然后，我们希望 Redis 在命令处理函数中自动管理资源和内存，而这可以简单地通过调用 `RedisModule_AutoMemory(ctx)` 来完成。

接下来，我们开始访问 Redis 的键，也就是我们示例中的两个 Redis 列表。Redis 为模块提供了两套用于访问 Redis 数据空间的 API。低级（low-level）API 可以访问 Redis 键并非常快速地操作数据结构。高级（high-level）API 则允许在模块的代码中调用 Redis 命令并获取结果，这与 Lua 脚本访问 Redis 数据空间的方式非常类似。在此案例中，我们使用了低级 API 中的 `RedisModule_OpenKey()` 来打开两个输入列表和目标哈希。该函数返回一个指向 `RedisModuleKey` 的指针，即 Redis 键的处理函数。然后，我们通过使用 `RedisModule_KeyType()` 来检查键类型以确保输入是列表或者是空类型（`REDISMODULE_KEYTYPE_EMPTY`）。

下一步是遍历两个列表并将值赋给哈希。这里，我们创建函数 `ListLPopRPush()` 来从列表中读取元素。该函数从列表左端弹出一个元素，随后又将其压回列表右端，最后将元素的值返回给调用者。我们使用 `RedisModule_ValueLength()` 来获取每个列表的长度，然后使用 `RedisModule_HashSet()` 来设置目标哈希中的值。当我们读取完其中一个列表中的所有元素时，遍历将停止，但是我们仍然需要从另一个列表（这个列表可能更长）中弹出和压入剩余元素，以使两个列表本身保持不变。

因为命令需要返回新创建哈希的长度，所以我们使用了 `RedisModule_Call()` 函数来调用 `HLEN` 命令。`RedisModule_Call()` 可被用来调用任意的 Redis 命令，并将结果作为一个指向 `RedisModule_CallReply` 的指针返回。`RedisModule_CallReplyInteger()` 将结果的值作为一个整数返回，然后这个整数被传给 `RedisModule_ReplyWithLongLong()` 并最终返回给命令的调用者。

`RedisModule_Call()` 支持可变长参数。其中，第一个参数是模块上下文，第二个参数是命令名称（以 `null` 结尾的 C 语言字符串），第三个参数是命令参数的格式说明符。因为参数可能源于不同类型的字符串：以 `null` 为结尾的 C 语言子字符串，在命令实现中从 `argv` 参数中接受的 `RedisModuleString` 对象，等等。在我们的示例中，`HLEN` 的参数来自于 `argv` 中接收到的 `RedisModuleString`，所以我们只需要将 `s` 放入格式说明符中即可。其余的参数是命令参数。

以下是格式说明符的完整列表：

- `c --`: 指向以 `null` 结尾的 C 语言字符串的指针。
- `b --`: C 语言缓冲区，需要两个参数，指向 C 语言字符串的指针和长度 `size_t`。
- `s --`: 在 `argv` 中接收到的 `RedisModuleString` 或其他 Redis 模块 API 返回的 `RedisModuleString` 对象。
- `l --`: 长整型。
- `v --`: `RedisModuleString` 对象数组。
- `!` --: 这个描述符只是告诉函数将命令复制到从实例和 AOF。从参数解析的角度来看，这个描述符会被忽略。

`RedisModule_ReplicateVerbatim(ctx)` 表示我们的命令会一字不差地传播给 Redis 的从实例。

例如，如果我们在一个 Redis 主实例上执行 `MYMODULE.ZIP myhash field values`，那么它的从实例也将一字不差地接收到这条命令。

11.3.4 更多细节

读者可能已经注意到，`RedisModule_OnLoad()` 也接受一个参数数组和参数个数作为该函数的参数。这些是可以在加载模块时指定的模块参数，如 `MODULE LOAD mymodule arg1 arg2 arg3`。当我们希望模块在根据传递的不同参数具有不同的表现时，这很有用。

11.3.5 相关内容

- 有关 Redis 模块 API 的完整参考，请参阅：<https://redis.io/topics/modules-api-ref>。
- Redis 模块 API 的官方介绍，请参阅：<https://redis.io/topics/modules-intro>。
- Dvir Volk 撰写的《编写 Redis 模块》，请参阅：<https://redislabs.com/blog/writing-redis-modules/>。
- 有一个名为 `RedisModuleSDK` 的库给 Redis 模块开发人员提供了一些实用的功能和宏，请参阅：<https://github.com/RedisLabs/RedisModulesSDK>。

第 12 章

Redis 生态系统

在本章中，我们将学习下列案例：

- Redisson 客户端。
- Twemproxy。
- Codis——一个基于代理的高性能 Redis 集群解决方案。
- CacheCloud 管理系统。
- Pika——一个与 Redis 兼容的 NoSQL 数据库。

12.1 本章概要

在数据库世界中，Redis 已经逐步成为被广泛使用的成熟开源软件。此外，还有很多基于 Redis 的开源项目。在本章中，我们将介绍 Redis 生态系统中的一些流行项目。

Redisson 客户端是一个具有很多扩展功能的 Java 客户端。它通过将 Redis 用作存储后端，减轻了开发人员实现分布式对象和服务的负担。

由 Twitter 开发的 Twemproxy 是一个同时支持 Redis 和 Memcached 协议的代理。它经常被用作 Redis 数据分片解决方案，也能够减少到后端 Redis 服务器的连接数量。

与 Twemproxy 类似，Codis 是另一个集中式的高性能代理。通过将请求路由到代理背后的 Redis 实例，Codis 提供了一个易于使用并可用于生产环境的分片解决方案，且数据分片管理无需开发人员付出额外努力。

管理大量的 Redis 实例会让人非常头疼。Sohu TV 开发的 CacheCloud 项目可以用于解决这个问题。从将新实例部署，到线上实例监控，CacheCloud 为几乎所有的 Redis 操作提供了一个开箱即用的在线管理解决方案。

虽然近年来物理内存的成本已经降低了很多，但使用固态硬盘作为 Redis 的数据存储后端还是很有可能节省成本的。奇虎 360 的 Pika 项目是一个实现了上述想法的兼容 Redis 接口的存储服务。

12.2 Redisson 客户端

Redisson 是一个增强的 Redis Java 客户端，它为使用 Redis 提供了一种更便捷的方式。Redisson 提供了一系列分布式对象和服务，能够简化使用 Redis 设计和实现大型分布式系统的难度。

图 12.1 展示了 Redisson 的主要功能。

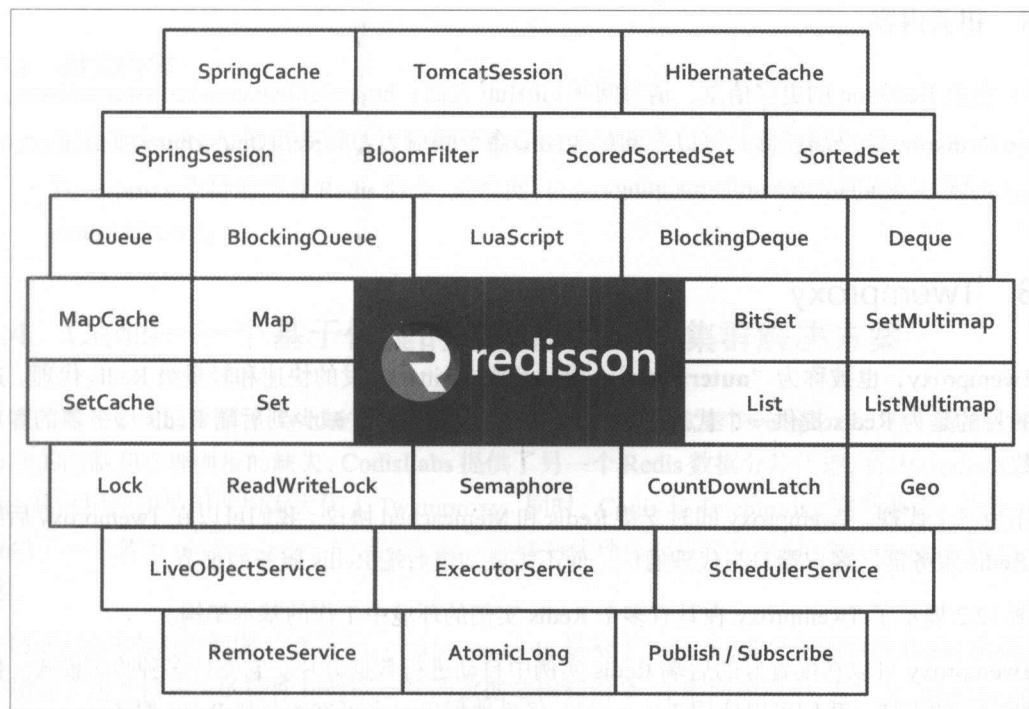


图 12.1 Redisson 的主要功能

Redisson 基于 Java NIO 中的 Netty 框架。它不仅可以在数据库驱动程序层上作为扩展的 Redis 客户端，还可以提供更高级的功能。诸如 hash、list、set、string、Geo 和 HyperLogLog 等原生的 Redis 数据类型，被封装成了易于使用的 Java 数据结构或对象（Map、List、Set、Object Bucket、Geospatial Bucket 和 HyperLogLog）。

此外，Redisson 还支持分布式数据类型，如 `Multimap`、`LocalCachedMap` 和 `SortedSet`。在 Redisson 库中，还实现了分布式锁、`MultiLock`、`ReadWriteLock`、`FairLock`、`RedLock`、`Semaphore` 和 `CountDownLatch` 对象。简而言之，在使用 Redis 构建分布式系统时，Redisson 是一个很有用的库。

通过 Redisson 库中的分布式环境工具，Redisson 还为不同的用例提供了分布式服务，如 `RemoteService`、`ExecutorService` 和 `SchedulerService`。

Redisson 也可以被看作是基于内存的数据网格。Redisson 节点是一个独立的任务处理节点，可以作为系统服务运行并自动加入 Redisson 集群。

Redisson 的目标是实现 Redis 用户关注点的分离；这样，开发人员可以将重点放在数据建模和应用程序的逻辑上。在 Redisson 的 *Redis 命令映射表*（*Redis commands mapping table*）的帮助下，我们可以很容易地从其他 Redis 客户端迁移到 Redisson。

12.2.1 相关内容

- 有关 Redisson 的更多信息，请参阅其 GitHub 页面：<https://github.com/redisson/redisson>。
- Redisson 客户端迁移时可以参阅的 Redis 命令映射表：<https://github.com/redisson/redisson/wiki/11.-Redis-commands-mapping>。

12.3 Twemproxy

Twemproxy，也被称为“**nutcracker**”，是一个由 Twitter 开发的快速和轻量级 Redis 代理。这个项目的目的是为 Redis 提供一个代理和数据分片的解决方案，并减少到后端 Redis 服务器的客户端连接数。

作为一个代理，Twemproxy 同时支持 Redis 和 Memcached 协议。我们可以在 Twemproxy 后配置多个 Redis 服务器。客户端只与代理通信，而不需要知道后端 Redis 服务的细节。

图 12.2 展示了 Twemproxy 在具有多个 Redis 实例的环境中工作的基本架构。

Twemproxy 可以在配置好的后端 Redis 实例中自动进行数据分片。它支持多种哈希模式。借助一致性哈希的支持，我们可以使用 Twemproxy 轻松地配置一个可靠的分片 Redis 服务。

Twemproxy 还可以配置为在失败时禁用后端节点，并在一段时间后重试。

使用 Twemproxy 并不意味着存在单点故障的问题。实际上，我们可以为同一组后端服务器运行多个 Twemproxy 实例，以防止单点问题出现。

Twemproxy 也存在一定的局限性。它并不支持所有的 Redis 命令；例如，`PUB/SUB` 及事务命令是不支持的。此外，为 Twemproxy 添加或删除后端 Redis 节点并不方便。首先，如果不重启 Twemproxy，

配置就不会生效。其次，虽然 Twemproxy 支持一致性哈希，但在添加或删除 Redis 节点后，数据不会自动重新平衡。

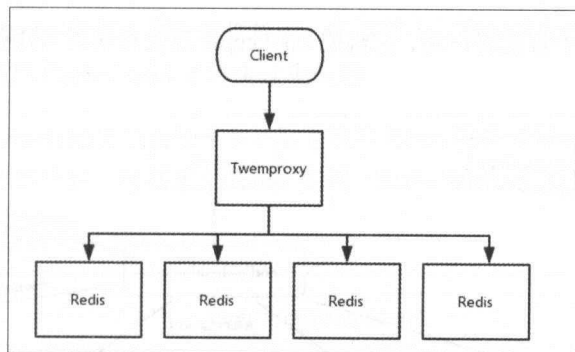


图 12.2 Twemproxy 的基本架构

12.3.1 相关内容

- 有关 Twemproxy 的更多信息，请参阅其 GitHub 页面：<https://github.com/twitter/twemproxy>。
- Twemproxy 支持的所有 Redis 命令，请参阅：<https://github.com/twitter/twemproxy/blob/master/notes/redis.md>。

12.4 Codis——一个基于代理的高性能 Redis 集群解决方案

在本章中，我们介绍了 Twitter 开发的基于代理的分片解决方案 Twemproxy。为了解决水平伸缩性方面的限制和管理面板的缺失，CodisLabs 提供了另一个 Redis 数据分片代理，称为 Codis。Codis 的性能、高可用性和易用性均大大优于 Twemproxy。同时，Codis 与 Twemproxy 完全兼容。此外，Codis 还提供了一个名为 `redis-port` 的便利工具，能够与代理一起进行从 Redis/Twemproxy 到 Codis 的迁移。

Codis 的基本架构如图 12.3 所示。`codis-server` 是基于 `redis-3.2.8` 分支的特殊 Redis 实例，并在其中增加了用于槽相关的操作和数据迁移的数据结构和指令。在前面的图中，有三个 `codis-server`，一个主实例和两个从实例。

`codis-group` 是作为部分数据集工作的一组 `codis-server`。在 Codis 集群中，数据集通过 CRC32 算法被划分为 1024 个槽。Codis 引入了 `codis-group` 的概念，每个组包含一个主实例和至少一个从实例。Redis Sentinel 用于管理 `codis-server` 的故障迁移。

`codis-proxy` 是主要的代理服务，为客户端连接实现了 Redis 协议。对于同一个生产集群，可以部署多个 Codis 代理，这些代理的状态是可以同步的。

codis-dashboard 是一个集群管理工具，可用于 codis-server 和 codis-group 的添加和删除。

jodis-client 是一个智能客户端，它监控 Zookeeper 以获得实时可用的代理，并以轮转的方式发送命令。

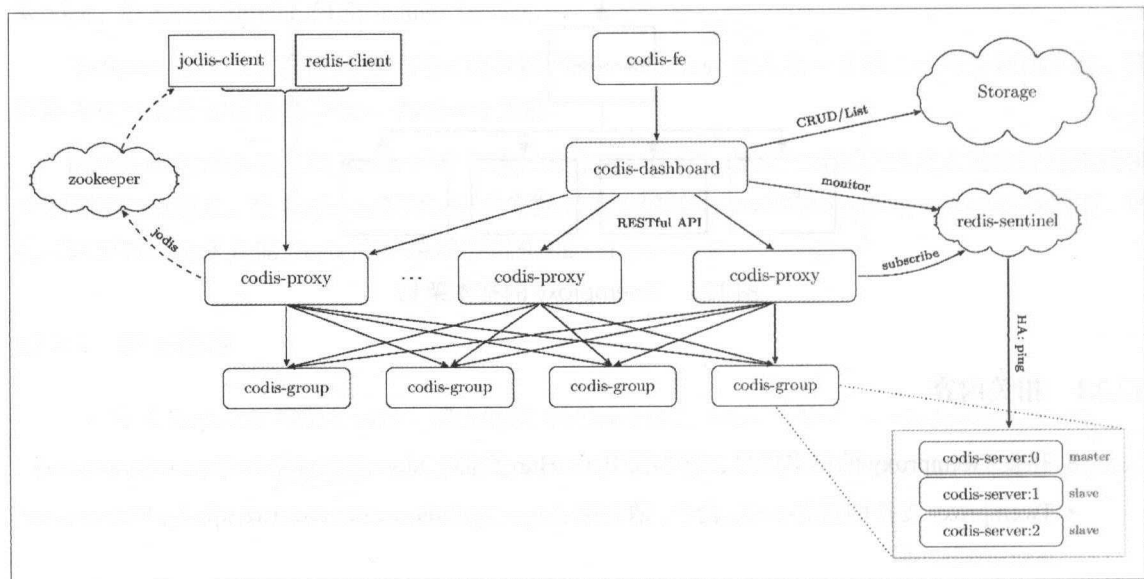


图 12.3 Codis 的基本架构

鉴于我们在前几章和本章中介绍了多种 Redis 数据分片解决方案。下面，就让我们来比较一下这些方案，以便让读者对各种解决方案更适合具体的场景建立基本的认知：

产品 \ 功能	Codis	Tewmproxy	Redis Cluster
重新分片无需重启集群	支持	不支持	支持
Pipeline	支持	支持	不支持
多键操作是否支持 hash tags	支持	支持	支持
重新分片期间支持多键操作	支持	不支持	不支持
Redis 客户端支持性	任意客户端	任意客户端	客户端必须支持 Clster 协议

12.4.1 相关内容

- 有关 Codis 的更多细节，请参阅其 GitHub 页面：<https://github.com/CodisLabs/codis>。
- 有关 Redis 迁移工具 redis-port 的更多细节，请参阅其 GitHub 页面：<https://github.com/CodisLabs/redis-port>。

12.5 CacheCloud 管理系统

部署和管理大量的 Redis 实例非常麻烦且耗时。在本节中，我们将介绍一个由搜狐（NASDAQ: Sohu）TV 开发的开源项目 CacheCloud 来解决这类问题。

CacheCloud 是一个 Redis 的集中管控平台，用于执行 Redis 服务交付、性能监控和故障诊断。它支持多种类型的 Redis 体系结构，包括单个 Redis 实例、Redis Sentinel 和 Redis 集群。

CacheCloud 的架构如图 12.4 所示。

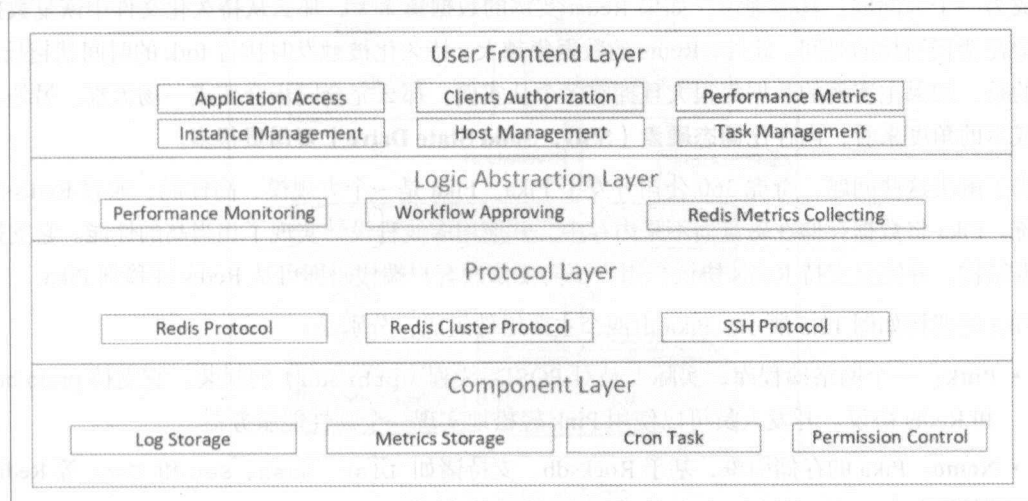


图 12.4 CacheCloud 的架构

CacheCloud 提供了下列优秀的功能：

- **服务交付 (Service Provisioning)**：希望在应用程序中使用 Redis 的用户可以轻松地应用 Redis 数据服务。单个 Redis 实例、Redis Sentinel 高可用和 Redis 集群都被支持，并且在访问应用被批准后能够自动部署。
- **性能监控**：管理员可以很容易地在一个仪表板上获取由 CacheCloud 管理的所有 Redis 实例的重要性能指标。
- **问题报警**：如果由 CacheCloud 管理的 Redis 实例中发生任何问题，用户将会收到通知。
- **配置/管理操作管理**：几乎所有常见的管理操作和配置变更都可以通过 CacheCloud 进行，从而消除通过终端手动进行操作的风险。
- **客户端访问管理**：提供了一个特殊的 Java 客户端来帮助用户收集 Redis 客户端的指标，以便在无需额外工作的情况下进行故障诊断。

12.5.1 相关内容

- 有关 CacheCloud 的更多细节，请参阅其 GitHub 页面：<https://github.com/sohutv/cachecloud>。
- CacheCloud 的主页，请参阅：<https://cachecloud.github.io/>。

12.6 Pika——一个与 Redis 兼容的 NoSQL 数据库

通过前面的介绍，我们了解到，如果一个 Redis 实例的数据集太大，那么主从复制和持久化很可能会成为一个大问题。具体地说，如果 Redis 实例的数据集太大，那么从持久化文件中恢复数据集总是会耗费相当长的时间。此外，Redis 的数据集越大，持久化被触发时执行 fork 的时间就越长。最糟糕的是，如果主实例的数据集很大且拥有多个从实例，那么完全同步会变成一场灾难。另外，从存储成本的角度来看，内存比固态硬盘（SSD，Solid State Drive）要昂贵很多。

为了解决这些问题，奇虎 360 公司开发了 Pika。Pika 是一个大规模、高性能、兼容 Redis 的存储系统。Pika 将数据存储在磁盘而不是内存中，并使用多线程设计实现了相当高的性能。它支持多种数据结构，并完全支持 Redis 协议。用户不需要修改客户端代码即可从 Redis 迁移到 Pika。

Pika 的架构如图 12.5 所示。Pika 由四个主要组件组成，分别是：

- **Pink**：一个网络编程库，实际上是对 POSIX 线程（pthread）的封装。它支持 proto buffer 和 Redis 协议。开发人员可以使用 Pink 轻松地实现一个高性能服务器。
- **Nemo**：Pika 的存储引擎，基于 Rocksdb，支持诸如 List、Hash、Set 和 Zset 等 Redis 数据类型。
- **Binlog**：由索引和偏移检查点组成的顺序日志。在 Pika 中，主实例和从实例之间的同步是通过 Binlog 完成的。
- **Working thread**：Pika 使用多个工作线程来进行读写操作，线程安全由底层的 Nemo 引擎保证。

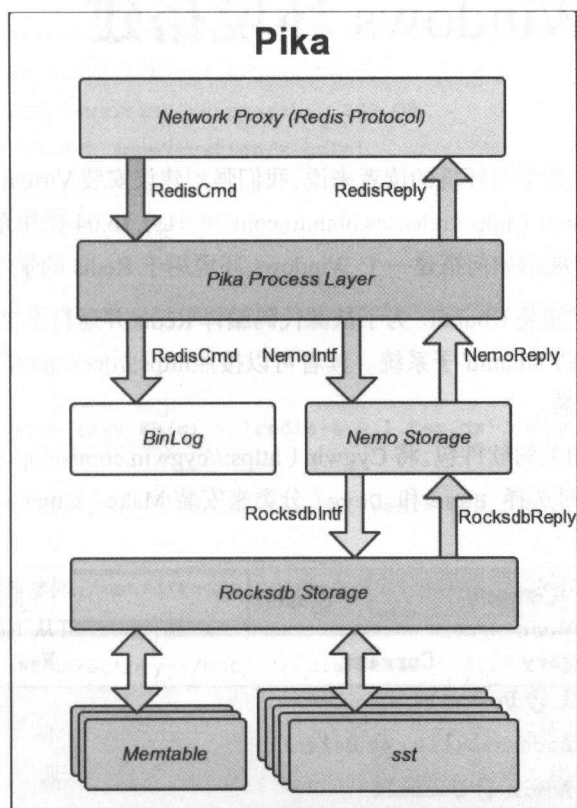


图 12.5 Pika 的架构

12.6.1 相关内容

- 有关 Pika 的更多细节，请参阅其 GitHub 页面：<https://github.com/Qihoo360/pika>。

附录 A Windows 环境搭建

对于使用 Windows 作为学习环境的读者来说,我们强烈建议安装 VirtualBox (<https://www.virtualbox>). 然后在虚拟机上安装 Ubuntu (<http://releases.ubuntu.com/16.04/>) 16.04 操作系统。如果读者想坚持使用 Windows, 那么本章将展示如何搭建一个 Windows 环境用于 Redis 的学习:

1. 在 Windows 10 上安装 Ubuntu。为了从源代码编译 Redis 并运行本书中的 Bash shell 脚本, 我们需要安装 Windows 10 的 Ubuntu 子系统。读者可以按照<https://docs.microsoft.com/en-us/windows/wsl/about>中的说明进行安装。

2. 安装 Cygwin 及相关的软件包。将 Cygwin (https://cygwin.com/setup-x86_64.exe) 安装到 `c:\tools\cygwin` 目录, 同时选择 Base 和 Devel 分类来安装 Make、Cmake、gcc 和 bash, 如图 1 所示。

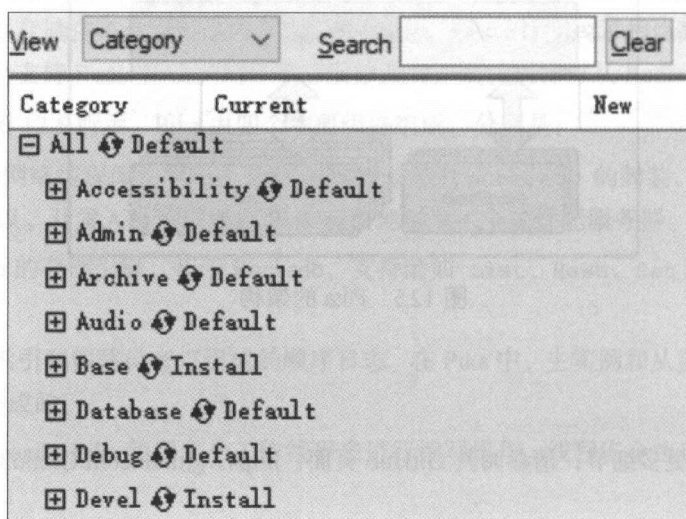


图 1 安装 Cygwin

3. 按如下的步骤编译并安装 Redis。

```
user@DESKTOP-JUTUKJS:~$ cd /mnt/c
user@DESKTOP-JUTUKJS:/mnt/c$ mkdir redis
user@DESKTOP-JUTUKJS:/mnt/c$ cd redis/
user@DESKTOP-JUTUKJS:/mnt/c/redis$ mkdir bin
user@DESKTOP-JUTUKJS:/mnt/c/redis$ mkdir conf
user@DESKTOP-JUTUKJS:/mnt/c/redis$ wget http://download.redis.io/releases/
redis-4.0.1.tar.gz
```

```
--2018-01-23 10:52:25-- http://download.redis.io/releases/redis-4.0.1.tar.gz
Connecting to 127.0.0.1:1080... connected.
Proxy request sent, awaiting response... 200 OK
Length: 1711660 (1.6M) [application/x-gzip]
Saving to: 'redis-4.0.1.tar.gz'

redis-4.0.1.tar.gz
100%[=====>] 1.63M 270KB/s
in 6.7s

2018-01-23 10:52:34 (249 KB/s) - 'redis-4.0.1.tar.gz' saved [1711660/1711660]

user@DESKTOP-JUTUKJS:/mnt/c/redis$ tar xzf redis-4.0.1.tar.gz

user@DESKTOP-JUTUKJS:/mnt/c/redis/redis-4.0.1$ make
cd src && make all
make[1]: Entering directory '/mnt/c/redis/redis-4.0.1/src'
CC Makefile.dep
...
user@DESKTOP-JUTUKJS:~/redis-4.0.1$ make PREFIX=/mnt/c/redis install
cd src && make install
make[1]: Entering directory '/home/user/redis-4.0.1/src'
CC Makefile.dep
...
make[1]: Leaving directory '/mnt/c/redis/redis-4.0.1/src'

user@DESKTOP-JUTUKJS:/mnt/c/redis$ cd ..
user@DESKTOP-JUTUKJS:/mnt/c/redis$ mkdir conf
user@DESKTOP-JUTUKJS:/mnt/c/redis$ cp redis-4.0.1/redis.conf conf/.

user@DESKTOP-JUTUKJS:~$ cd
user@DESKTOP-JUTUKJS:~$ ln -s bin /mnt/c/redis/bin
user@DESKTOP-JUTUKJS:~$ ln -s conf /mnt/c/redis/conf
```

4. 设置 C 语言集成开发环境 CLion。从<https://download.jetbrains.com/cpp/CLion-2017.2.2.exe>下载 C/C++ 集成开发环境 CLion，然后双击 setup 文件并按照提示完成安装。

5. 接下来，将 echodemo.tar.gz 解压到 c:\redis\redis-4.0.1 目录中。源代码是和本书

一起提供的（读者也可以在第 1 章开始使用 Redis 中找到源代码）。图 2 展示了对应的目录结构：



Name	Size	Type	Modified	Attr
.idea	12.5 KB	File Folder	Today 9:02	----
cmake-build-debug		File Folder	Yesterday 15:17	----
deps		File Folder	Yesterday 14:12	----
echodemo		File Folder	星期日 18:26	----
src		File Folder	星期日 18:26	----
tests		File Folder	2017/7/24 21:58	----
utils		File Folder	2017/7/24 21:58	----
.gitignore	376 bytes	File	Yesterday 10:52	-a---
00-RELEASENOTES	124 KB	File	Yesterday 10:52	-a---
BUGS	53 bytes	File	Yesterday 10:52	-a---
CMakeLists.txt	3.19 KB	文本文档	星期六 2:08	----
CONTRIBUTING	1.77 KB	File	Yesterday 10:52	-a---
COPYING	1.45 KB	File	Yesterday 10:52	-a---
INSTALL	11 bytes	File	Yesterday 10:52	-a---
Makefile	151 bytes	File	Yesterday 10:52	-a---
MANIFESTO	4.12 KB	File	Yesterday 10:52	-a---
README.md	20 KB	MD 文件	Yesterday 10:52	-a---
redis.conf	56.4 KB	CONF 文件	Yesterday 10:52	-a---

图 2 目录结构

6. 打开 CLion 并导入项目。注意不要覆盖 CMakeList.txt。

准备构建工具链，依次通过 Files->Settings->Build, Execution, Deployment->Toolchains 设置 Cygwin 环境，如图 3 所示。

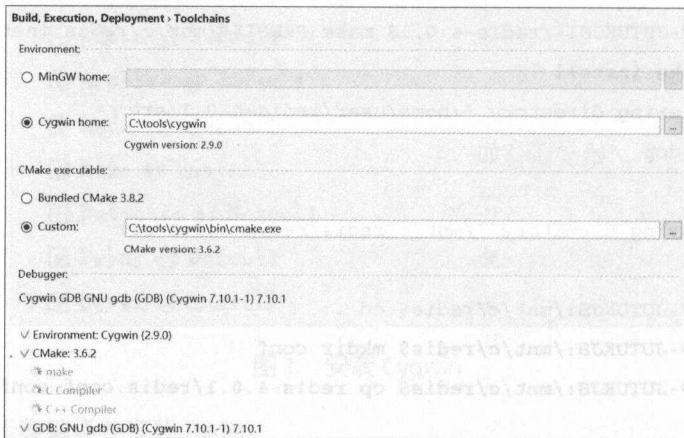


图 3 设置 Cygwin 环境

7. 在 CLion 中打开命令行终端，并创建一个名为 deps 的文件夹。忽略编译期间的警告：

```
C:\redis\redis-4.0.1>cd deps
```

```
C:\redis\redis-4.0.1\deps>make CFLAGS=-D_WIN32 CPPDEFS=-D_WIN32 hiredis lua  
linenoise
```

```
(cd hiredis && make clean) > /dev/null || true
```

```
...
make[1]: Leaving directory '/cygdrive/d/redis-4.0.1/deps/linenoise'
```

8. 使用 CLion 中的 Build All 构建所有可执行文件。点击 Build All 按钮，如图 4 所示：

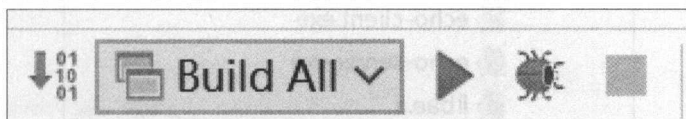


图 4 构建所有可执行文件

9. 忽略以下的警告并点击 Continue Anyway，如图 5 所示。

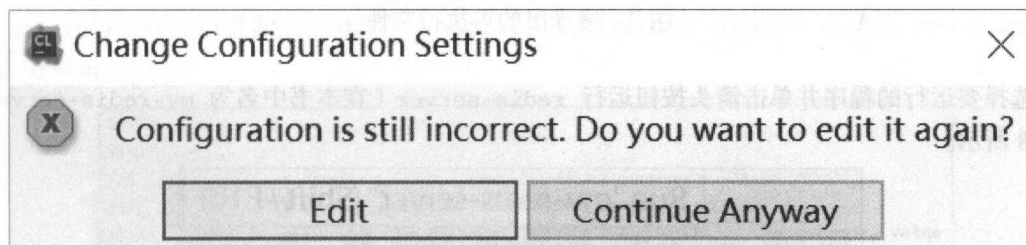


图 5 忽略警告

我们会发现示例代码开始编译，如图 6 所示。

```
[ 92%] Building C object CMakeFiles/my-redis-server.dir/src/t_string.c.o
[ 93%] Building C object CMakeFiles/my-redis-server.dir/src/t_zset.c.o
[ 94%] Building C object CMakeFiles/my-redis-server.dir/src/util.c.o
[ 96%] Building C object CMakeFiles/my-redis-server.dir/src/ziplist.c.o
[ 97%] Building C object CMakeFiles/my-redis-server.dir/src/zipmap.c.o
[ 98%] Building C object CMakeFiles/my-redis-server.dir/src/zmalloc.c.o
[100%] Linking C executable my-redis-server.exe
[100%] Built target my-redis-server
```

图 6 开始编译

至此，我们成功地编译了 demo 程序和 redis-server 的源代码。所有的可执行文件都位于 cmake-build-debug 文件夹下，如图 7 所示。

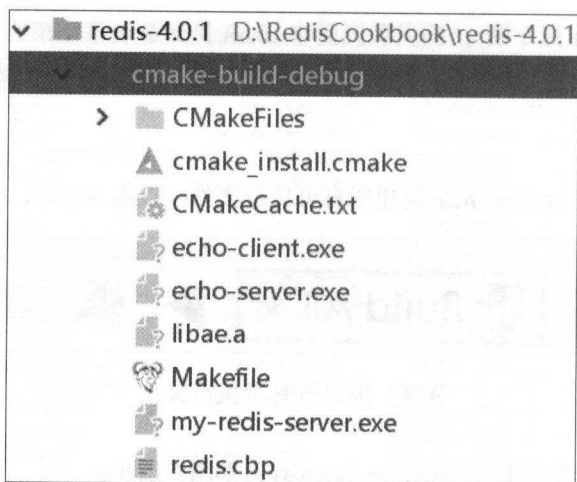
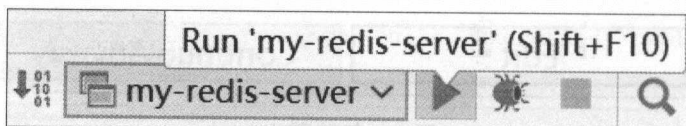


图 7 编译出的可执行文件

选择要运行的程序并单击箭头按钮运行 `redis-server`（在本书中名为 `my-redis-server`），如图 8 所示。

图 8 运行 `redis-server`

Redis 服务器将启动，如图 9 所示。

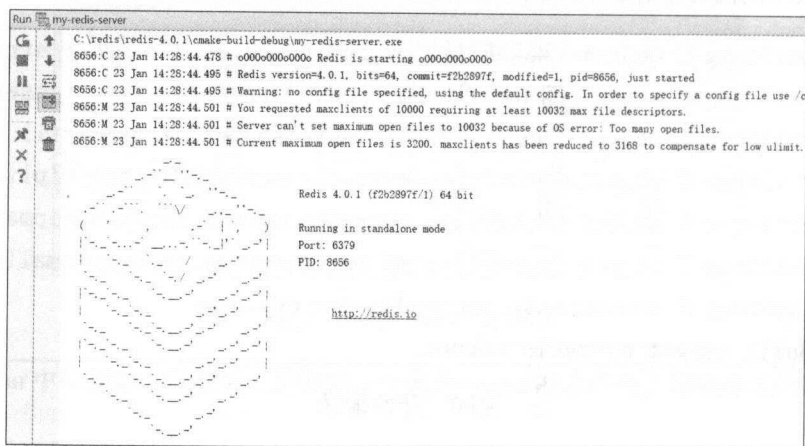


图 9 启动 Redis 服务器

此时，我们可以使用第 3 步中编译出的 `redis-cli` 连接到这个 `redis-server`。执行一些测

试，然后关闭 Redis 实例：

```
user@DESKTOP-JUTUKJS:/mnt/c/redis$ bin/redis-cli
127.0.0.1:6379> set foo bar
OK
127.0.0.1:6379> get foo
"bar"

127.0.0.1:6379> get foo
127.0.0.1:6379> SHUTDOWN
not connected>
```

我们还可以通过设置断点（例如，在 `server.c` 的 `main()` 函数中）来调试 `redis-server`，如图 10 所示。

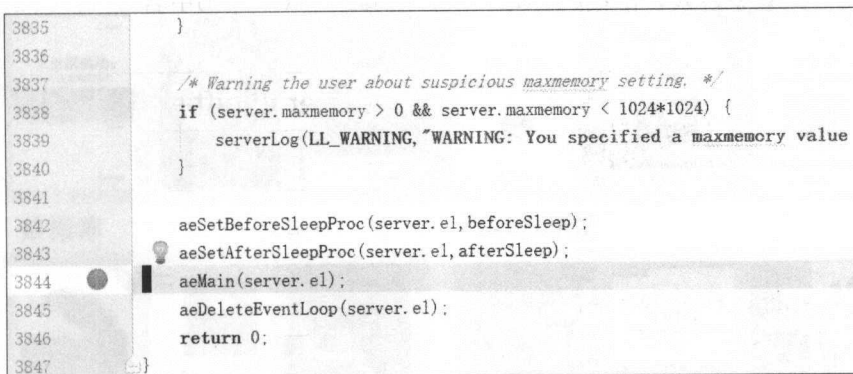


图 10 调试 `redis-server`

单击 `Debug` 按钮开始调试，如图 11 所示。

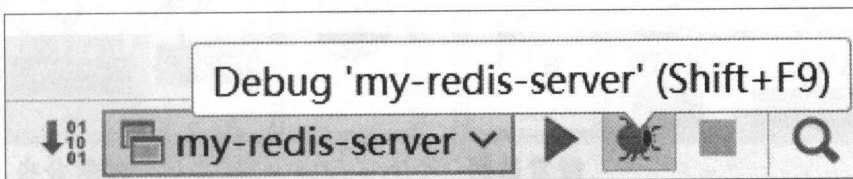


图 11 开始调试

随后进行调试即可，如图 12 所示。

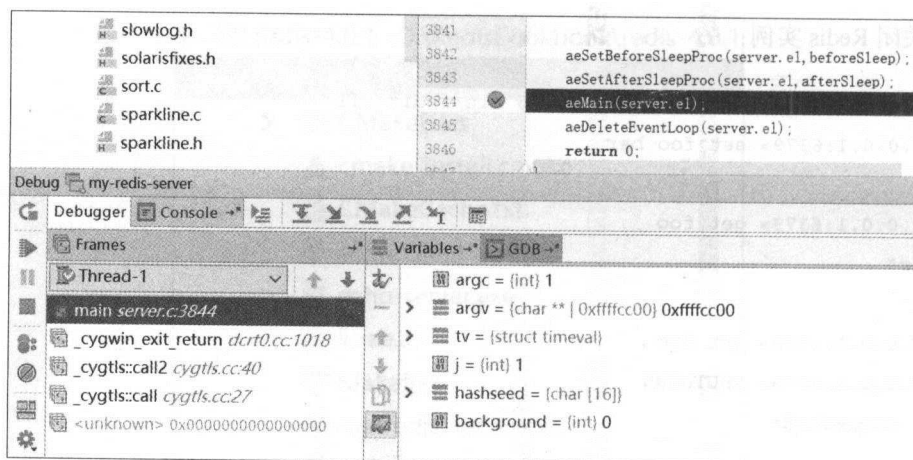


图 12 进行调试

非卖品！！严禁（售卖和上传互联网平台）！！
 仅供对书籍质量进行鉴定甄别！是否为购买正版实体书提供依据！！

电子工业出版社
 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
 http://www.phei.com.cn

Broadview
 www.broadview.com.cn

博文视点·IT出版旗舰品牌

博文视点精品图书展台

专业典藏



移动开发



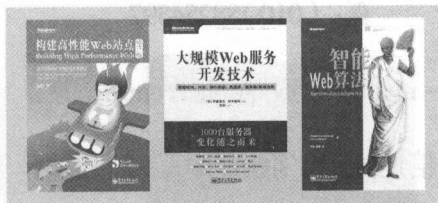
大数据·云计算·物联网



数据库



Web 开发



程序设计



软件工程



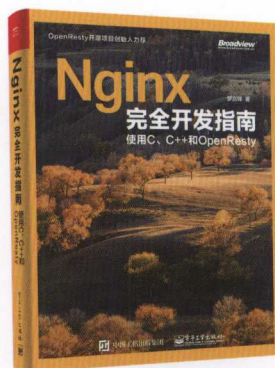
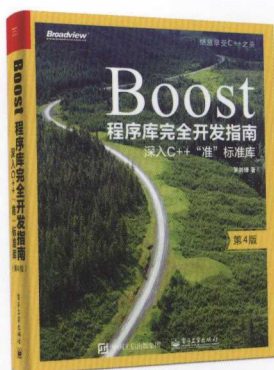
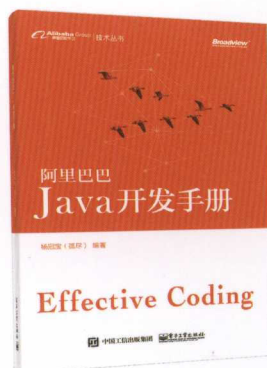
办公精品



网络营销



非卖品！！严禁（售卖和上传互联网平台）！！
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！



拒绝堆砌臃肿,支持纯正原创

出版事宜请关注  新浪微博 @ 半亩方塘 _
weibo.com

投稿邮箱: sxy@phei.com.cn

非卖品！！严禁（售卖和上传互联网平台）！！

仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

Broadview
www.broadview.com.cn

博文视点·IT出版旗舰品牌

技术凝聚实力·专业创新出版



专家评论

Redis已经成为开发运维人员的“标配”技术。本书语言精练、内容丰富，包含大量开发运维中的优化方案和案例，同时还全面介绍了Redis 4.0及其生态系统的内容，对于全面掌握Redis，这是一本不可多得的好书。

付磊 《Redis开发与运维》作者、阿里云数据库技术专家

认识鹏程几年，一直见证他对于Redis的付出，经常能看到他总结、收集国内外各种使用经验和业务场景，也非常热心地回答大家关于Redis的各种问题。他写作的这本书从Redis 4.x基本命令实战、复制、持久化到各种高可用集群方案，均做了详尽的讲解，同时也介绍了Redis的模块开发扩展的入门基础，并且包括Codis等多个主流Redis生态中的产品。感谢鹏程给大家带来的这本书，真诚推荐给喜欢Redis的用户，可作为日常案头参考。

刘奇 PingCAP 创始人&CEO、CodisLabs创始人

Redis是目前流行的缓存数据库，应用广泛。该书从基础使用与开发、高级特性、实战诊断等多个维度展开了详细的阐述，既适合入门也适合进阶，在技术实践的广度和深度上均有兼顾，是缓存数据库领域不可多得的匠心之作。

子嘉 阿里云数据库资深技术专家

通过本书，读者将学到：

- 安装和配置Redis实例
- Redis中的各种数据类型和命令
- 使用Redis构建客户端应用及大数据框架
- 在Redis中管理数据复制和持久化
- 在Redis中实现高可用性和数据分片
- 使用Redis模块扩展Redis
- 基准测试、调试，以及Redis中各种问题的故障诊断
- 了解Redis周边生态

上架建议：计算机与互联网>数据库



博文视点Broadview



@博文视点Broadview

ISBN 978-7-121-34081-9



9 787121 340819 >

定价：89.00元



责任编辑：孙学瑛

封面设计：侯士卿